Efficient, scalable consistency for highly fault-tolerant storage

GARTH GOODSON

August 2004

CMU-CS-04-111

Dept. of Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **AUG 2004**	2. REPORT TYPE	3. DATES COVERED **00-00-2004 to 00-00-2004**
4. TITLE AND SUBTITLE **Efficient, scalable consistency for highly fault-tolerant storage**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,Dept. of Electrical and Computer Engineering,Pittsburgh,PA,15213**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR’S ACRONYM(S)
		11. SPONSOR/MONITOR’S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT **see report**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**166**	

Keywords: survivable storage systems, Byzantine fault-tolerance, erasure codes

To my parents, for their unwavering support.

Abstract

Fault-tolerant storage systems spread data redundantly across a set of storage-nodes in an effort to preserve and provide access to data despite failures. One difficulty created by this architecture is the need for a consistent view, across storage-nodes, of the most recent update. Such consistency is made difficult by concurrent updates, partial updates made by clients that fail, and failures of storage-nodes.

This thesis demonstrates a novel approach to achieving scalable, highly fault-tolerant storage systems by leveraging a set of efficient and scalable, strong consistency protocols enabled by storage-node versioning. Versions maintained by storage-nodes can be used to provide consistency, without the need for central serialization, and despite concurrency. Since versions are maintained for every update, even if a client fails part way through an update, concurrency exists during an update, the latest complete version of the data-item being accessed still exists in the system—it does not get destroyed by subsequent updates. Additionally, versioning enables the use of optimistic protocols.

This thesis develops a set of consistency protocols appropriate for constructing block-based storage and metadata services. The block-based storage protocol is made space-efficient through the use of erasure codes and made scalable by offloading work from the storage-nodes to the clients. The metadata service is made scalable by avoiding the high costs associated with agreement algorithms and by utilizing threshold voting quorums. Fault-tolerance is achieved by developing each protocol in a hybrid storage-node fault-model (a mix of Byzantine and crash storage-nodes can be tolerated), capable of tolerating crash or Byzantine clients, and utilizing asynchronous communication.

Acknowledgements

I would first like to thank my advisor, Greg Ganger, without whose support and insight this work would not be possible. Next, I would like to thank Mike Reiter, whose patience and wealth of distributed systems knowledge guided us through this project. I would also like to thank the rest of my thesis committee members, Richard Golding, and Priya Narasimhan for their valuable comments, feedback, and time.

From my group, at the PDL, my warmest thanks goes to Jay Wylie, my research partner and friend. Without him, I would still be filling white boards with meaningless scribble. Next, I would like to thank Mike Abd-El-Malek, our newest PASIS member. He came up to speed on the complicated code-base in an amazingly short amount of time and provided invaluable help in implementing the query/update protocol. Finally, from my group, I would like to thank those that made my time at CMU a very enjoyable experience. Especially those from previous projects: John Strunk and Craig Soules; as well as Andy Klosterman, Jiri Schindler, Shuheng Zhou; and everyone else from my group: Steve Schlosser, John Griffin, Adam Pennington, Eno Thereska, Brandon Salmon. A special thanks goes to the PDL staff, especially to Karen Lindenfelser and Linda Whipkey for their fantastic organization and caring, positive attitude.

I would like to thank my family, especially my parents for providing their unending support. Last, but not least, I would like to thank my wonderful wife Vianey. I thank her for her patience and the sacrifices she made as she waited for me to finish my thesis. But, most of all, I would like to thank her for her support, friendship, and undying love.

I would like to thank the members and companies of the PDL Consortium (including EMC, Engenio, Hewlett-Packard, HGST, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433 and F30602–99–2–0539–AFRL, and by the Army Research Office, under agreement number DAAD19–02–1–0389.

Finally, I would like to thank Miguel Castro and Rodrigo Rodrigues for making the implementation of BFT publicly available.

Contents

1 Introduction 1

1.1 Problem definition 1

1.2 Thesis statement 3

1.2.1 Validation 3

1.3 Overview 4

1.3.1 Consistency protocols 4

1.3.2 Guiding assumptions 5

1.3.3 Applying the protocols to the PASIS storage system 6

1.4 Organization 8

2 Background and Related Work 9

2.1 Storage system overview 9

2.1.1 File service 10

2.1.2 Storage-system goals 12

2.2 Data encodings 13

2.2.1 RAID 13

2.2.2 Erasure-coding 14

2.3 Consistency semantics 15

2.3.1 Linearizability 15

2.3.2 Serializability 16

2.3.3 Tolerating benign faults 17

2.3.4 Tolerating Byzantine faults . . . 18
2.3.5 Metadata scalability . . . 19

3 Read/Write Block Protocol 23
3.1 System model . . . 24
3.2 Mechanisms . . . 26
3.2.1 Erasure codes . . . 27
3.2.2 Data-fragment integrity . . . 27
3.2.3 Write operation integrity . . . 28
3.2.4 Authentication . . . 30
3.3 Protocol . . . 30
3.3.1 Overview . . . 30
3.3.2 Pseudo-code . . . 31
3.4 Constraints . . . 36
3.4.1 Read classification . . . 36
3.4.2 Real repairable candidates . . . 37
3.4.3 Decodable repairable candidates . . . 37
3.4.4 Write termination . . . 37
3.4.5 Constraint summary . . . 38
3.5 Implementation . . . 38
3.5.1 Storage-node implementation . . . 38
3.5.2 Garbage collection . . . 39
3.5.3 Client implementation . . . 40
3.5.4 Erasure codes . . . 41
3.6 Evaluation . . . 42
3.6.1 Experimental setup . . . 42
3.6.2 Performance and scalability of PASIS protocol members . . . 42
3.6.3 Performance and scalability comparison with BFT . . . 46
3.6.4 Other results . . . 52

3.7 Discussion 53
3.7.1 Byzantine clients 53
3.7.2 Timestamps from Byzantine storage-nodes 53
3.7.3 Garbage collection 54
3.8 Summary 54

4 Read/Conditional Write Block Protocol 57
4.1 Overview 58
4.1.1 R/CW semantics 58
4.1.2 Read operation overview 60
4.1.3 CW operation overview 62
4.2 Mechanisms 63
4.2.1 Validating timestamps 63
4.2.2 Authenticators 64
4.3 Protocol 65
4.3.1 Read operation 65
4.3.2 CW operation 67
4.3.3 CW requests at storage-nodes 70
4.4 Constraints 75
4.4.1 Read classification rules 75
4.4.2 Real repairable candidates 76
4.4.3 Complete candidate—repairable candidate intersection 76
4.4.4 Read set intersection 77
4.4.5 CW termination 77
4.4.6 Constraint summaries 78
4.4.7 Improving the bounds on N and Q_{C} 79
4.4.8 Liveness 80
4.4.9 Safety 81
4.5 Protocol scalability 81

4.6 Discussion 82
4.6.1 Erasure codes 82
4.6.2 Invalid authenticators 83
4.6.3 Storage-nodes rejecting CW requests 84
4.7 Evaluation 85
4.8 Summary 87

5 Metadata Service 89
5.1 Overview 91
5.2 Metadata operations 92
5.2.1 Query operations 92
5.2.2 Update operations 94
5.2.3 Multi-object operations 95
5.2.4 Summary 97
5.3 Improving efficiency 98
5.3.1 Breaking large objects into blocks 99
5.3.2 Object synchronization 99
5.4 PASIS metadata objects 100
5.4.1 Attribute objects 100
5.4.2 Directory objects 101
5.4.3 Lock (lease) objects 102
5.4.4 Authorization objects 103
5.5 Storage-system implementation 103
5.5.1 Metadata operations 104
5.5.2 PMD metadata-nodes 105
5.5.3 PMD clients 108
5.5.4 Storage service 110
5.6 Evaluation 112
5.6.1 Experimental setup 112

5.6.2 Cryptography performance 112
5.6.3 PMD micro-benchmarks 113
5.6.4 PMD service macro-benchmarks 116
5.6.5 PASIS file system: SSH-build 122
5.7 Summary 124

6 Conclusions and Future Work 125
6.1 Conclusion 125
6.2 Contributions 126
6.3 Future directions 127

Bibliography 128

A Read/write safety and liveness 141
A.1 Proof of safety 141
A.1.1 Safety guarantees 141
A.1.2 Proof 142
A.2 Proof of liveness 146
A.2.1 Liveness guarantees 146
A.2.2 Proof 146

Figures

1.1 High-level architecture for survivable storage. 2

2.1 Example 2-of-5 erasure coding scheme. 15

3.1 Example of cross checksum generation for 5 data-fragments. 27

3.2 Example of a “poisonous write” by a Byzantine client. 28

3.3 Verification of a validating cross checksum at read time. 29

3.4 Pseudo-code for storage-node *S*. 32

3.5 Client-side write operation pseudo-code. 33

3.6 Client-side read operation pseudo-code. 34

3.7 Illustration of constraint on *N*. 39

3.8 Computational cost of erasure coding. 44

3.9 Mean response time vs. total failures tolerated. 48

3.10 Protocol cost breakdown. 49

3.11 Throughput vs. number of clients ($b = 1$). 50

4.1 Examples illustrating the *conditioned-on chain*. 60

4.2 Read requests and candidate classification. 61

4.3 Client read and classification pseudo-code. 66

4.4 Example of repair requiring a barrier-write. 68

4.5 Client-side CW operation pseudo-code. 69

4.6 Example of a barrier-write. 70

4.7 Reception of a CW at storage-node *S*. 71

4.8 Validation of a CW request at storage-node *S*. 72
4.9 Logical timestamps returned from classification. 73
4.10 Validation postulates. 75
4.11 Mean response time vs. total failures 86

5.1 Architecture of scalable storage systems. 90
5.2 Client query pseudo-code. 93
5.3 Client multi-obj repair pseudo-code. 96
5.4 Example of multi-object repair. 97
5.5 Example metadata operations. 98
5.6 PASIS storage system. 104
5.7 Concurrency experiments. 115
5.8 Response time distributions of a fault-free and multiple fault-induced runs. 117
5.9 Postmark throughput vs. client load. 119
5.10 Postmark run time vs. total failures tolerated (t). 120
5.11 Throughput of threshold quorum system vs. Δ. 122
5.12 Run time of SSH build for Linux, PMD, and PMD + PS. 123

Tables

3.1 Protocol constraint summary 38

3.2 Client and storage-node computation costs. 45

5.1 Implemented PMD service operations. 105

5.2 Micro-benchmarks of NFS operations. 114

5.3 Threshold quorum experiment parameters ($b = t = 1$). 121

1 Introduction

1.1 Problem definition

Fault-tolerant storage systems (e.g., Petal [Lee and Thekkath 1996], Myriad [Chang et al. 2002], SwiftRAID [Long et al. 1994], and Cheops [Amiri et al. 2000a]) spread data redundantly across a set of storage-nodes in an effort to preserve and provide access to data despite failures. Figure 1.1 illustrates the abstract architecture of a fault-tolerant, or survivable, distributed storage system. In these types of systems it is common to break the system (at least logically) into two components or services: a metadata service; and a data service. To access or update a data-item, a client must obtain metadata about the data-item (e.g., where and how it is stored) before it is able to access the data-item itself. In order for a storage-system to tolerate failures, both the data and metadata must be duplicated across a set of storage-nodes. Thus, an update is only complete once it has completed successfully at a subset of the storage-nodes. While this scheme provides access to data-items and their metadata even when subsets of the storage-nodes have failed, it does create the difficulty of maintaining a consistent view, across the storage-nodes, of the most recent update. In decentralized systems, this is problem is exacerbated since, due to the lack of a central serialization point, updates may not be issued to the same subset of storage-nodes as are being read. Without consistency across the set of storage-nodes, data loss is possible or even likely.

Although protocols exist for achieving such consistency, they generally fall short in a number of areas including fault-tolerance, efficiency, and scalability. The easiest solution to this problem is to introduce a point of serialization. This is typically done by serializing

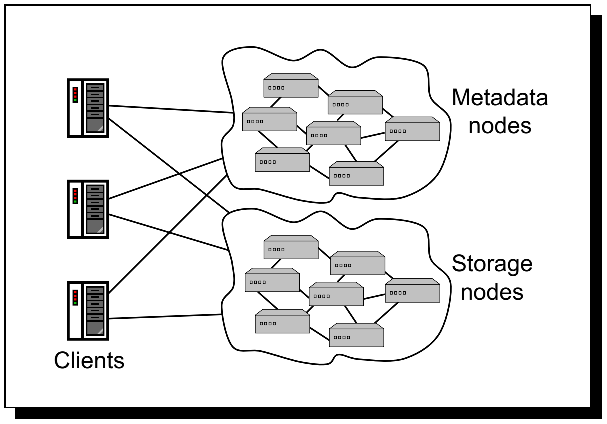


Figure 1.1: **High-level architecture for survivable storage.** *Spreading data and metadata redundantly across storage-nodes improves its fault-tolerance. Clients write and (usually) read data from multiple storage-nodes and may contact multiple storage-nodes to perform metadata operations.*

requests through a primary. However, this typically reduces the scalability of the system and requires additional protocols to tolerate the failure of the primary. Other more complicated protocols exist, however they generally require a significant amount of overhead in the common case of little or no concurrency. Most studies of distributed storage systems (e.g., [Baker et al. 1991; Noble and Satyanarayanan 1994]) indicate that concurrency is uncommon (i.e., writer-writer and writer-reader sharing occurs in well under 1% of operations). As well, many protocols do not scale in terms of messaging or protocol overhead, or storage-node CPU utilization as number of faults tolerated increases.

This thesis demonstrates a novel approach to achieving scalable, highly fault-tolerant (the ability to tolerate more than a single fault) storage systems by leveraging a set of efficient and scalable, strong consistency protocols enabled by storage-node versioning. Versioning storage-nodes keep a version of every update they receive (for some period of time). These versions can be used to provide consistency, without the need for central serialization, and despite concurrency. Since versions are maintained for every update, even if a client fails part way through an update, or a reader performs a query during an update, the latest complete version of the data-item being accessed still exists in the system—it does not get destroyed by subsequent updates. The problem with versioning becomes one of the client locating the version it is interested in. The consistency protocols developed in this thesis all use logical time as a means of naming versions. Additionally, versioning enables the use of optimistic protocols. Since older versions are not overwritten by new updates, there is no need to lock the data-item before performing an update. However, in these protocols, concurrency may require the client to perform extra work to find the set of versions in which they are interested (e.g., those that comprise the latest complete update).

In particular, this thesis develops a set of consistency protocols appropriate to building block-based storage and metadata services. The block-based storage protocol is made space-efficient through the use of erasure codes and made scalable by offloading work from the storage-nodes to the clients. The metadata storage protocol is made scalable by avoiding the high costs associated with agreement algorithms and by utilizing threshold voting quorums. Fault-tolerance is achieved by developing each protocol in a hybrid storage-node fault-model (a mix of Byzantine and crashed storage-nodes can be tolerated), capable of tolerating crash or Byzantine clients, and utilizing asynchronous communication.

1.2 Thesis statement

Versioning storage-nodes enable the design of a set of scalable, efficient consistency protocols that provide a foundation for constructing scalable, highly fault-tolerant, distributed storage systems.

1.2.1 Validation

This work is validated through the design and evaluation of three consistency protocols that have been enabled by versioning storage-nodes. More precisely:

(1) It develops and demonstrates a read/write block storage consistency protocol that enables highly fault-tolerant storage through the use of erasure coded data and versioning storage-nodes. Its correctness is shown through proof sketches.

(2) It develops and demonstrates a read/conditional-write block protocol that allows for stronger read–modify–write consistency semantics. Additionally, tradeoffs between tolerating Byzantine clients and erasure coding, as well as tradeoffs between tolerating Byzantine storage-nodes and liveness are discussed.

(3) It extends the read/conditional-write block protocol to support operations on multiple, arbitrary objects and implements a scalable metadata service based upon this and the read/write protocol.

(4) It evaluates a distributed file system that utilizes the scalability and fault-tolerance of the developed consistency protocols in terms of the number of faults tolerated, the maximum throughput the system can sustain, and its performance in degraded operation modes (i.e., with concurrency and faults).

1.3 Overview

1.3.1 Consistency protocols

First, the Read/Write protocol (R/W) is developed. It provides strong consistency and fault-tolerance for read/write block storage. Block storage-systems (e.g., SCSI, fibre-channel) provide the backbone for most current storage solutions.

The R/W protocol works roughly as follows. To perform a write, clients write time-stamped fragments to at least a write threshold of storage-nodes. Storage-nodes keep all versions of fragments they are sent. To perform a read, clients fetch the latest fragment versions from a read threshold of storage-nodes. The client determines whether the fragments comprise a consistent, complete write, based on timestamp ordering; usually, they do. If they do not, additional fragments or historical fragments are fetched, or repair is performed, until a consistent, complete write is observed. Only in cases of failures (storage-node or client) or read-write concurrency is additional overhead incurred to maintain consistency.

The second protocol, the Read/Conditional Write protocol (R/CW), extends the R/W protocol by only allowing write operations to complete if the object has not changed since the last time it was read. Thus, the R/CW protocol is able to provide stronger consistency semantics—similar to read–modify–write, rather than just read–write semantics.

Finally, a Query/Update protocol (Q/U) is developed. It is very similar to the R/CW protocol but adds a few optimizations and extensions. Most notably, it provides strict serializability of arbitrary operations through the use of replicated state machines.

These protocols are developed in detail, evaluated individually, and used as a basis for building a fault-tolerant, scalable storage-system.

1.3.2 Guiding assumptions

These consistency protocols achieve efficiency and scalability via a combination of optimistic operation, versioning, and quorum-style redundancy. As such, there are a number of assumptions that guide the use of versioning and optimism. As well, there are a number of assumptions in the system model for which these protocols are designed.

Optimism and versioning

The scalability features of quorums are well-known [Malkhi et al. 2000; Naor and Wool 1998], however the use of versioning and optimism is guided by three high-level assumptions.

First, we assume that client failures within the duration of an access protocol (on the order of milliseconds) should be rare. That is, while we design to tolerate client failures (indeed, arbitrary ones; see below), our protocols optimistically presume they will not occur, and exploit this assumption heavily in order to improve throughput when it holds.

Second, we assume that comprehensive object versioning at each metadata node is efficient. Previous studies have shown that versioning nodes can offer performance that is typically within 10% of a non-versioning node [Strunk et al. 2000]. As well, modern disks have the capacity required to version objects comprehensively [Strunk et al. 2000; Soules et al. 2003].

Third, we assume that objects exported through the protocols, designed properly, will experience low access concurrency. Most file system studies conclude that file sharing is rare. For example, our R/CW objects support conditional write operations that update multiple objects atomically. This, in turn, permits us to utilize fine-grained metadata objects, which reduces access concurrency for these objects. Thus, a separate attribute object can be maintained for each file, rather than including file attributes in directory objects.

System model

There are a number of system model assumptions that hold for all protocols developed. The system model is more formally described in Section 3.1, but can be summarized as follows.

Each data-item is hosted by a static number of storage-nodes; i.e., once the data-item has been created, the set of storage-nodes on which that data-item can exist is fixed. There are an arbitrary number of clients in the system. Both storage-nodes and clients may suffer Byzantine faults [Lamport et al. 1982].

All protocols are developed within an asynchronous model of time (i.e., no assumptions are made about message transmission delays or execution rates). Channels are assumed to be point-to-point, authenticated, and adhere to finite duplication and fair loss properties [Aguilera et al. 2000]; see 3.1 for a complete description of the system model.

1.3.3 Applying the protocols to the PASIS storage system

The PASIS storage system is layered above the consistency protocols described in the previous subsection. It is split into two components: the PASIS Storage (PS) service and the PASIS metadata (PMD) service. The storage service may be implemented using either the Read/Write or the Read/Conditional Write protocol. The metadata service is is built upon the Query/Update protocol since the consistency semantics required for metadata is more stringent than that for data. Additionally, the Q/U protocol provides an interface to atomically perform arbitrary operations on multiple metadata objects.

The R/W protocol underlies the PS service. It provides block granularity read/write access to data objects. Data objects are variable length data containers named by a unique object identifier. The R/W protocol allows for the use of space-efficient data encodings. To demonstrate that our protocol is efficient in practice, we compare its performance to BFT [Castro and Liskov 2001; 2002], the Byzantine fault-tolerant replicated state machine implementation that Castro and Liskov have made available [Castro and Rodrigues 2003]. Experiments show that the PS scales better than BFT in terms of network utilization at the server and in terms of work performed by the server. Experiments also show that response times of PASIS and BFT are comparable. Additionally, experiments show that the response time graphs of the PASIS R/W prototype are flat as the number of faults tolerated is scaled up.

Two types of metadata objects are implemented: attributes and directories. Attributes objects exist for both directory objects and for files. The attributes map directly to typical UNIX file permissions. Directory objects hold multiple directory entries. Each directory entry stores the names and access information for the files and directories stored within the storage system. The access information specifies how the named object can accessed. If the named object is a file, the access information is specific to the PS service implementation (e.g., where the file is located, the encoding of the file, etc.).

The PMD service is evaluated in the context of a complete file system implemented as a NFS server. It can use either the PS service to store data, or it can be configured to store data locally in its local file system. When storing data locally, experiments show that the PMD service's throughput scales as load (number of NFS servers) is increased and response time only gradually increases as the number of faults tolerated is scaled up. As well, experiments show that the performance degrades gracefully when concurrency is introduced, even at very high concurrency levels. Finally, when the PS service is used in conjunction with the PMD service in a configuration capable of tolerating a single Byzantine fault, the run time of an OpenSSH build is within a factor of two of a non-fault tolerant user-level NFS server.

1.4 Organization

The remainder of this thesis is organized as follows. Chapter 2 describes background and related work. It is broken into a discussion of atomic read/write objects (or registers) that pertains to block based storage and a discussion of systems/protocols capable of providing consistency and fault-tolerance for operations performed on arbitrary objects. Chapter 3 develops the R/W protocol for block-based storage. The system model, constraints on the number of storage-nodes, and the implementation and evaluation of the protocol are described. Chapter 4 describes the R/CW protocol for block based storage. The protocol is developed similarly to the R/W protocol. Chapter 5 extends the R/CW protocol to provide consistency for operations performed over arbitrary objects (i.e., the Q/U protocol). As well, the chapter describes the design and implementation of the PASIS storage system that utilizes both the Q/U protocol and the R/W to provide strong consistency, fault-tolerance, and scalability to its clients. The storage system is then evaluated in terms of a distributed NFSv3 storage system. The last chapter, Chapter 6, concludes and provides future directions for this work. Finally, a set of appendices provide proofs of safety for the consistency protocols developed within.

2 Background and Related Work

This chapter describes background and related work related to the construction of scalable and fault-tolerant distributed storage systems. First, the components that comprise a storage system are described. Second, data encoding schemes that can be used to improve space-efficiency are introduced. Third, consistency semantics and protocols for tolerating benign and Byzantine faults are described. Fourth, and lastly, work related to the scalability of metadata services is discussed.

2.1 Storage system overview

Traditionally, disk-based storage systems have been built around a centralized monolithic disk array or mainframe. While these systems have been shown to provide good reliability and performance, they have a number of weaknesses. First, the hardware is highly customized and very expensive to build. Second, these systems are hard to scale to very large sizes. Third, the range of faults they are able to handle is limited (e.g., benign single, or possibly double, disk failures).

This thesis describes protocols that can be used to build a Byzantine fault-tolerant, decentralized storage architecture to help solve these problems. First, by tolerating Byzantine faults cheaper, off-the-shelf, components can be used since hardware and software bugs can be masked by the fault-tolerance provided by the underlying storage protocols. Second, these systems are more scalable in that the addition of new storage-nodes yields improvements in the capacity, throughput or fault-tolerance of the service. Third, fault-tolerance is gained by designing the storage protocols to withstand arbitrary (Byzantine) failures of clients and a limited number of metadata-nodes, and by requiring no timing (synchrony) assumptions for correctness. However, in this type of architecture, there is no centralized control, making it difficult to provide consistency in the face of faults and concurrency.

2.1.1 File service

This work focuses on developing protocols that can be used to construct a decentralized, fault-tolerant file based storage-system. Traditional file systems are comprised of both metadata and data services. The data service is responsible for storing file data, while the metadata service stores data about how and where the file data is stored (e.g., block pointers within inodes), as well as other metadata that describes the file (e.g., attributes, access control information, etc.). Metadata is often stored within the data service and is accessed by recursing through a set of structures rooted at a well-known location.

In these systems, fault-tolerance for both the data and metadata can be obtained by distributing the data service in a fault-tolerant manner. Frangipani [Thekkath et al. 1997] is an example of this type of system. It is a distributed file system that is built above a virtual disk interface exported by Petal [Lee and Thekkath 1996] and a distributed lock service. Petal can tolerate one or more disk or storage-node failures, as long as the majority of the storage-nodes are up and communicating, and as long as at least one replica of each data-item remains.

Other systems explicitly separate the metadata service from the data service. For example, NASD [Gibson et al. 1998] demonstrated that by separating metadata access from data access greater scalability could be achieved at a lower cost. Instead of forcing all operations through a centralized file server, NASD eliminated the file server from the data flow path by allowing clients to directly access the data storage-nodes. To increase fault-tolerance the centralized metadata server can be distributed as a fault-tolerant service. For example, Farsite [Adya et al. 2002] utilizes a Byzantine fault-tolerant agreement protocol (BFT [Castro and Liskov 1998a]) to protect the integrity of its metadata, while allowing file data to be stored on a user's desktop machine.

Consistency semantics

Consistency semantics can differ for data versus metadata. Most block based data services, disk drives being the most common, expect whole block updates (i.e., an entire block is always overwritten). On the other hand, metadata services often allow arbitrary data regions to be updated independently (e.g., a single directory entry may be altered within a directory).

For block updates, it is sufficient to support *read–write* update semantics. Read-write update semantics make no guarantees about the value of the data block between the time the block was read and later written. These semantics are sufficient for block stores, since consistency is guaranteed on a block-level and blocks are usually read and written as atomic units. The PASIS read–write (R/W) protocol is described in Chapter 3 and provides the consistency semantics required for block based storage.

In order to support consistent updates to metadata, metadata objects (e.g., directories) require update operations that *modify* their existing contents, rather than blindly overwriting their previous contents; otherwise, their integrity may not be preserved. *Read–modify–write* semantics guarantee that the data region has not been modified between a read and a successive write operation to the same data region. It is also necessary to support atomic updates across multiple objects (e.g., when renaming or moving files from one directory to another). Metadata services are often built upon protocols that provide consistent access to objects that can be manipulated through arbitrary operations (i.e., not just read and write operations). In the PASIS metadata service, the underlying *read–conditional write* (R/CW) protocol is described in Chapter 4, while the query/update (Q/U) protocol, that extends the R/CW protocol to provide replicated state-machine semantics, and the metadata service itself is described in Chapter 5.

This thesis describes a set of protocols that provide the consistency necessary to implement fault-tolerant data and metadata services.

2.1.2 Storage-system goals

One central goal in the design of storage systems is to simultaneously provide efficiency, scalability, and fault-tolerance. Current storage systems, and their underlying protocols, fall short in one or more of the following areas:

– High fault-tolerance: To provide access to data in the event of multiple client and/or server failures (in the case of both crash and Byzantine faults), as opposed to tolerating only a single failure as can be handled by most other distributed storage systems. First, data must be spread redundantly across the set of storage-nodes. Second, no central points of failure should exist. This can be achieved by using decentralized consistency protocols with no single points of failure.

– Strong consistency: To provide strong consistency in the face of failures (of clients or servers) and concurrent operations (e.g., read-write concurrency, write-write concurrency). In decentralized storage systems, where data is spread across multiple storage-nodes, it is usually important to ensure that readers and writers always see a consistent view of data, especially in the face of concurrency and failures. Although this is a goal that we want of our storage systems, not all applications require strong consistency. As well, the consistency semantics required of block level storage versus metadata is different. At the metadata level, it is important to offer consistency of metadata operations which may span multiple objects.

– Efficiency and scalability: To provide scalable access to data and low overheads in the common case of fault-free, concurrency-free operation. Many protocols exist for providing consistency, however there is generally a significant amount of overhead regardless of the state of the system (e.g., when concurrency and faults do not exist). Current protocols are also generally inflexible in dealing efficiently with different fault models (e.g., crash vs. Byzantine failures, number of failures to be masked) and system models (e.g., synchronous vs. asynchronous); i.e., they are designed to work for a single fault and timing model. Although designing for the worst case generally provides support for many system and failure model assumptions, efficiency and scalability are always limited to that of the worst case environment.

2.2 Data encodings

A common data distribution scheme used in distributed storage systems is replication, in which a writer stores a replica of the new data-item value at each storage-node to which it sends a write request. Since each storage-node has a complete instance of the data-item, the main difficulty is identifying and retaining the most recent instance. It is often necessary for a reader to contact multiple storage-nodes to ensure that it sees the most recent instance. Examples of distributed storage systems that use this design include Harp [Liskov et al. 1991], Petal [Lee and Thekkath 1996], BFS [Castro and Liskov 1998a], and Farsite [Adya et al. 2002].

Alternately, more space-efficient encoding schemes can be used. This section provides an overview of some of the more well-known schemes, such as RAID, and some other more general, more space-efficient erasure coding schemes. With these schemes, reads require fragments from multiple servers. Moreover, to decode the data-item, the set of fragments read must correspond to the same write operation; thus write–write concurrency can be problematic.

2.2.1 RAID

In order to increase the performance of disk subsystems, data can be striped across a set of disks. However, as the number of disks in each stripe is increased, the likelihood of a single disk dying and the probability of data loss increases. In 1988 Paterson, et al. [Patterson et al. 1988] designed Redundant Arrays of Inexpensive Disks (RAID) to overcome these reliability challenges. They solved the problem by storing redundant data in the form of parity on one or more of the disks in the array.

At the present time there are a number of different RAID levels. The most common are RAID 0, 1, and 5. Combinations of these levels also exist (e.g., RAID 10). RAID 0 is the simplest, increasing performance by striping data across a set of devices, so that they can be read and written in parallel. However, RAID 0 provides no extra redundancy.

RAID 1 provides mirroring of data onto two devices. This scheme can tolerate a single device failure, however it pays a huge cost in storage capacity—only half of the space is usable to store data. RAID 5 uses parity with striping to improve space-efficiency. Like in RAID 0, data is striped across a set of devices. A parity code is calculated simply by performing an XOR over the blocks within each stripe and is stored on separate device. In RAID 5 the parity block is rotated among the set of the storage devices.

2.2.2 Erasure-coding

The use of erasure codes can greatly improve the space-efficiency of replicating data. Erasure codes were originally developed for communication channels in the networking community and are sometimes known as forward-error correcting codes. Erasure codes encode a data-item into a set of fragments and have the property that any subset of a certain size can be used to reconstruct the original data-item. They have the nice property that they can tolerate m simultaneous failures with only m extra data-fragments. RAID schemes can typically only support $m = 1, 2$; i.e., they can only tolerate a single or double disk failure.

In this work the focus is on systematic threshold erasure codes in which any m of the n encoded data-fragments can decode the data-item. The first m data-fragments are stripes of the data-item. The remaining $n - m$ code-fragments are generated using polynomial interpolation within a Galois field. As such, each fragment is $\frac{1}{m}$ the total data size. Thus, the total size-blowup is $\frac{n}{m}$. Replication can be thought of a subset of these m-of-N erasure codes, as could the different RAID schemes; $m = 1$ for replication. Examples of such codes are Reed-Solomon codes [Berlekamp 1968], secret sharing [Shamir 1979], information dispersal (IDA) [Rabin 1989], short secret sharing [Krawczyk 1994], and “tornado” codes [Luby et al. 2001]. The tradeoff in using erasure codes over RAID like schemes is the performance cost in generating the code fragments.

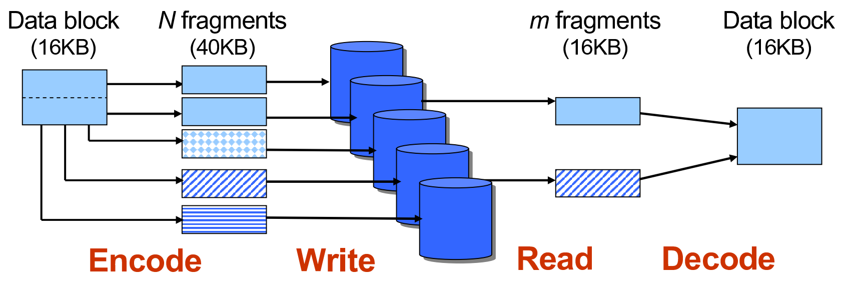


Figure 2.1: **Example 2-of-5 erasure coding scheme.** *An example 2-of-5 erasure code is shown (i.e., $m = 2$, $N = 5$). The first 2 fragments are stripe fragments, while the last 3 fragments are code fragments. Each fragment is $\frac{1}{2}$ the total data size. The total storage overhead is $\frac{N}{m}$ for N fragments.*

An example 2-of-5 erasure code scheme is shown in Figure 2.1. The original data-item is striped into 2 fragments, with 3 code fragments being generated. Each fragment is written to a storage-node. Any 2 fragments can be used to decode the original data-item.

There exists much prior work (e.g., [Agrawal and El Abbadi 1990; Herlihy and Tygar 1987; Mukkamala 1994]) that combines erasure coded data with quorum systems to improve the confidentiality and/or integrity of data along with its availability. However, these systems do not provide consistency (i.e., a synchronization mechanism is required) and do not cope with Byzantine clients. Concurrently with our own work, Frølund et al. [Frølund et al. 2004] have developed a decentralized protocol for linearizable erasure coded read/write registers that utilize a variant of threshold-quorums.

2.3 Consistency semantics

To provide reasonable semantics, storage systems must guarantee that readers see consistent data-item values.

2.3.1 Linearizability

The linearizability of operations is desirable for block-based read-write storage. Since linearizability is only defined for single object operations, it is not suitable for describing multi-object operations that are sometimes required for metadata updates. Linearizability is described by Herlihy and Wing in [Herlihy and Wing 1990]. Operations are *lineariz-*

able if their return values are consistent with an execution in which each operation is performed instantaneously at a distinct point in time between its invocation and completion. Frølund et al. [Frølund et al. 2004] have recently developed a block-based protocol that provides a variant of linearizability they term *strict linearizability* [Aguilera and Frolund 2003], in which an operation that crashes either takes effect within some limited time frame or not at all.

The R/W protocol, described in Chapter 3, tolerates Byzantine faults of any number of clients and a limited number of storage nodes while implementing linearizable [Herlihy and Wing 1990] and wait-free [Herlihy 1991] read-write objects. As well, the R/CW protocol, described in Chapter 4 also implements linearizable objects. In this protocol, linearizability is adapted appropriately for Byzantine clients, and wait-freedom as in the model of Jayanti et al. [Jayanti et al. 1998]. Since operations performed by Byzantine clients have no clear start time, they are excluded from the set of linearizable operations.

2.3.2 Serializability

The consistency semantic of serializability can pertain to multi-object operations that are required for updating metadata objects atomically. Traditionally serializability has been defined for transactions within database systems. A sequence of transactions are *serializable* if their outcome is equivalent to some sequential execution of the individual transactions [Papadimitriou 1979]. Strict serializability extends serializability to ensure that transactions already in the history in serial order (i.e., they have completed), remain in that relative order. This provides a consistency semantic similar to that of linearizability.

A serializable execution satisfies the *ACID* properties [Haerder and Reuter 1983] (i.e., atomicity, consistency, isolation, and durability). The serializability of transactions can be ensured through a number of techniques. Typical techniques include: two-phase locking [Gray et al. 1976], in which locks are acquired in one phase and released in a separate phase; optimistic concurrency control [Kung and Robinson 1981], in which operations within a transaction are performed optimistically, with no locking, and validation of serializability is done at commit time; and timestamp ordering [Bernstein et al. 1980], in these protocols timestamps are used to order operations. The query/update protocol described in Chapter 5 implements metadata objects that conform to strict serializability.

2.3.3 Tolerating benign faults

To provide operation atomicity, concurrency and client failures must be tolerated. A challenge introduced by concurrency and client failures is partially completed write operations. Partial writes arise from both write operations in progress and write operations that never completed (e.g., failed client).

Common approaches to dealing with partial writes in non-Byzantine-tolerant systems are two-phase commit [Gray 1978] and repair (write-back). Two-phase commit provides failure atomicity (although such protocols may block). Three phase commit protocols [Skeen and Stonebraker 1983] provide failure atomicity without blocking by utilizing failure detectors and/or recovery mechanisms. Alternately, many non-Byzantine-tolerant systems (e.g., Harp [Liskov et al. 1991] and Petal [Lee and Thekkath 1996]) serialize their actions through a primary storage-node, which becomes responsible for completing the update.

A common approach to dealing with concurrency is to suppress it, either via leases [Gray and Cheriton 1989] or optimistic concurrency control [Kung and Robinson 1981]. Ensuring operation atomicity in the face of Byzantine failures of clients requires additional work.

An alternate approach to handling both partial writes and concurrency is to have the data stored on storage-nodes be immutable [Reed and Svobodova 1980; Reed 1983]. By definition, this eliminates the difficulties of updates for existing data. In doing so, it shifts the problem up one level; an update now consists of creating a new data-item and modifying the relevant name to refer to it. Decoupling the data-item creation from its visibility simplifies both, but making the metadata service fault-tolerant often brings back the same issues.

For example, SWALLOW [Reed and Svobodova 1980] utilizes immutable object version logs (or histories) ordered by *pseudo time* to guarantee strong consistency (i.e., serializability) of arbitrary sets of read/write operations performed on a set of objects. As well, the Amoeba File Server [Mullender 1985] utilizes immutable data versions to implement optimistic concurrency control, such that the file system is always kept in a consistent state. More recently, peer-to-peer systems (e.g., Past [Rowstron and Druschel 2001] and CFS [Dabek et al. 2001]), Farsite, and the archival portion of OceanStore [Kubiatowicz et al. 2000] use immutable versions of data to simplify serialization of access to data. Other systems, such as Ivy [Muthitacharoen et al. 2002], use immutable version logs containing both data and metadata, however Ivy does not implement strong consistency guarantees for its metadata (or data) in this fashion.

Frølund et al. [Frølund et al. 2004] recently developed a decentralized consistency protocol for erasure coded data. Their algorithm relies on a quorum construction similar to threshold-quorums that they call "m-quorums" (any two quorums intersect in m processes). They utilize client generated timestamps to totally order updates and utilize server-side logs to track outstanding requests. Also, as described earlier, their protocol provides a variant of linearizability they call strict linearizability [Aguilera and Frolund 2003]. However, their protocol does allow for read and write operations to abort, as such they forgo strong liveness guarantees.

2.3.4 Tolerating Byzantine faults

Byzantine fault-tolerant protocols for implementing read-write objects using quorums are described in [Herlihy and Tygar 1987; Malkhi and Reiter 1997; Martin et al. 2002; Pierce 2001]. Of these related quorum systems, only Martin et al. [Martin et al. 2002] achieve linearizability in our fault model, and this work is also closest to ours in that it uses a type of versioning. In our protocol, a reader may retrieve fragments for several versions of the data-item in the course of identifying the return value of a read. Similarly, readers in [Martin et al. 2002] "listen" for updates (versions) from storage-nodes until a complete write is observed. Conceptually, our approach differs by clients reading past versions, versus listening for future versions broadcast by servers. In our fault model, especially in consideration of faulty clients, our protocol has several advantages. First, our protocol works for erasure-coded data, whereas extending [Martin et al. 2002] to erasure coded data appears nontrivial. Second, ours provides better message efficiency: [Martin et al. 2002] involves a $\Theta(N^2)$ message exchange among the N servers per write (versus no server-to-server exchange in our case) over and above otherwise comparable (and linear in N) message costs. Third, ours requires less computation, in that [Martin et al. 2002] requires digital signatures by clients, which in practice is two orders of magnitude more costly than the cryptographic transforms we employ. Advantages of [Martin et al. 2002] are that it tolerates a higher fraction of faulty servers than our protocol, and does not require servers to store a potentially unbounded number of data-item versions. Our prior analysis of versioning storage, however, suggests that the latter is a non-issue in practice [Strunk et al. 2000], and even under attack this can be managed using a garbage collection mechanism we describe in Section 3.6.

A metadata service, like any deterministic service, can be implemented in a survivable fashion using state machine replication [Schneider 1990], whereby all operations are processed by server replicas in the same order (*atomic broadcast*). While this approach supports a linearizable, Byzantine fault-tolerant implementation of *any* deterministic object, such an approach cannot be wait-free [Fischer et al. 1985; Herlihy 1991; Jayanti et al. 1998]. Instead, such systems achieve liveness only under stronger timing assumptions, such as synchrony (e.g., [Cristian et al. 1995; Pittelli and Garcia-Molina 1989; Shrivastava et al. 1992]) or partial synchrony [Dwork et al. 1988] (e.g., [Castro and Liskov 2002; Kihlstrom et al. 2001; Reiter and Birman 1994]), or probabilistically (e.g., [Cachin et al. 2001]). An alternative is Byzantine quorum systems [Malkhi and Reiter 1997], from which our protocol inherits techniques (i.e., our protocol can be considered a Byzantine quorum system that uses the threshold quorum construction). Protocols for supporting a linearizable implementation of any deterministic object using Byzantine quorums have been developed (e.g., [Malkhi et al. 2001]), but also necessarily forsake wait-freedom to do so. Additionally, most Byzantine quorum systems utilize digital signatures which are computationally expensive.

2.3.5 Metadata scalability

This section describes work related to building scalable metadata services. Numerous previous systems have focused on horizontally scaling data services through the addition of storage-nodes to obtain high data throughput. However, most systems either utilize a centralized metadata service or partition metadata across a set of servers such that each piece of metadata is handled by a single metadata server. The former approach is limited in its ability to scale, and both approaches render a metadata operation susceptible to a fault or compromise of the server responsible for it.

For example, NASD [Gibson et al. 1998] and Swift [Cabrera and Long 1991] centralize access to a metadata server. IBM's Storage Tank [Menon et al. 2003] and Lustre [Braam 2004] replace the central metadata server with a cluster of servers, partitioning metadata across the servers while supporting server fail-over. Likewise, some systems partition certain metadata structures (e.g., the manager map in xFS [Anderson et al. 1996] and the lock table in Frangipani [Thekkath et al. 1997]). Other systems make use of distributed protocols that communicate among the metadata servers to provide a replicated, fault-tolerant metadata service (e.g., Paxos [Lamport 1998] in Frangipani and BFT [Castro and Liskov 2002] in Farsite [Adya et al. 2002]) and OceanStore [Kubiatowicz et al. 2000]. Lastly, in some systems the storage-devices export interfaces directly to the client that provide serialized access to the device (e.g., device-served locks in GFS [Soltis et al. 1996] and base storage transactions by Amiri et al. [Amiri et al. 2000b]).

Survivable file systems have typically focused on the use of Byzantine fault-tolerant replication to protect the metadata service (e.g., [Deswarte et al. 1991]). Modern examples such as Farsite [Adya et al. 2002], OceanStore [Kubiatowicz et al. 2000], and BFS [Castro and Liskov 2002] employ state machine replication [Schneider 1990] for this purpose. While a powerful paradigm, state machine replication suffers from fundamental scaling limitations. First, all service nodes process all requests, so update throughput generally does not improve with additional nodes. Second, since the message complexity in their underlying agreement protocols is $\theta(n^2)$ with n replicas, the effect of adding nodes can be to degrade metadata update throughput. As such, adding replicas to the service group is of limited value: the throughput of read-only operations may improve, but the throughput of update operations at best would remain constant.

Consequently, to allow the metadata service in, e.g., Farsite to scale, the file system name-space is partitioned across multiple metadata services [Adya et al. 2002]. However, partitioning the name-space introduces another difficulty, namely implementing metadata operations atomically across replica groups, particularly in a manner resilient to Byzantine servers and clients. We are aware of no metadata service implementation that achieves this.

Our protocols employ a different paradigm that permits better load-balancing of requests across servers and linear-or-better message complexity per client request, and thus better ability to scale throughput as new servers are added. Rather than partitioning the name-space, we implement all metadata operations with a single replica group, and scale via lighter-weight access protocols than those implementing state machine replication. In the spirit of quorum protocols [Malkhi et al. 2000; Naor and Wool 1998], our approach permits clients to involve only a subset of servers in each operation (with no server-to-server communication). In particular, each read or update operation need only execute on a subset of metadata-nodes. Since all metadata operations are served in the same replica group, our approach can implement any metadata operation atomically. Thus, our metadata objects are, in effect, replicated state machines.

Extending our conditional write protocol to send update operations and to receive operation results, rather than sending and receiving whole objects, is efficient for objects with large state (e.g., directory objects). The optimistic nature of the conditional write protocol distinguishes it from other Byzantine quorum protocols. However, the protocol does not achieve the lower bound on N for implementing a Byzantine-tolerant replicated state-machine (i.e., $N \geq 4b+1$ [Malkhi and Reiter 1998a]).

The protocols developed in this thesis are most closely related to threshold-quorum systems (i.e., a majority voting system [Gifford 1979; Thomas 1979]), though our approach offers opportunities for exploring use of richer quorum constructions (e.g., [Malkhi and Reiter 1998a; Malkhi et al. 2000]). In a threshold-quorum system, the load [Naor and Wool 1998; Malkhi et al. 2000] on each storage-node is at least one half. This means that each storage-node must execute requests for at least one half of the operations applied to objects it hosts. True Byzantine quorum systems [Malkhi and Reiter 1998a] scale better than the one half bound. If Byzantine quorum construction techniques such as the M-Path construction [Malkhi et al. 2000] are employed, then the lower bound on load is $\Omega(\sqrt{\frac{b}{N}})$.

3 Read/Write Block Protocol

This chapter describes and evaluates a new consistency protocol that operates in an asynchronous environment and tolerates Byzantine failures of clients and storage-nodes. The protocol supports a hybrid failure model in which up to t storage-nodes may fail: $b \leq t$ of these failures can be Byzantine and the remainder can be crash. The protocol also supports use of m-of-n erasure codes (i.e., m-of-n fragments are needed to reconstruct the data), which usually require less network bandwidth (and storage space) than full replication [Weatherspoon and Kubiatowicz 2002; Wylie et al. 2000].

Briefly, the protocol works as follows. To perform a write, a client determines the current logical time (by querying a subset of the storage-nodes) and then writes time-stamped fragments to at least a threshold quorum of storage-nodes. Storage-nodes keep all versions of fragments they are sent until garbage collection frees them. To perform a read, a client fetches the latest fragment versions from a threshold quorum of storage-nodes and determines whether they comprise a completed write; usually, they do. If they do not, additional and historical fragments are fetched, and repair may be performed, until a completed write is observed.

The protocol gains efficiency from five features. First, the space-efficiency of m-of-n erasure codes can be substantial, reducing communication overheads significantly. Second, most read operations complete in a single round trip: reads that observe write concurrency or failures (of storage-nodes or a client write) may incur additional work. Most studies of distributed storage systems (e.g., [Baker et al. 1991; Noble and Satyanarayanan 1994]) indicate that concurrency is uncommon (i.e., writer-writer and writer-reader sharing occurs in well under 1% of operations). Failures, although tolerated, ought to be rare. Third, incomplete writes are replaced by subsequent writes or reads (that perform repair), thus preventing future reads from incurring any additional cost; when subsequent writes do the fixing, additional overheads are never incurred. Fourth, most protocol processing is performed by clients, increasing scalability via the well-known principle of shifting work from servers to clients [Howard et al. 1988]. Fifth, the protocol only requires the use of cryptographic hashes, rather than more expensive cryptographic primitives (e.g., digital signatures).

This chapter describes the protocol in detail, develops bounds for thresholds in terms of the number of failures tolerated (i.e., the protocol requires at least $2t+2b+1$ storage-nodes), and provides a proof sketch of its safety and liveness. The protocol requires at least $2t+2b+1$ storage-nodes (i.e., $4b+1$ if $t=b$). It also describes and evaluates its use in a prototype storage system called PASIS [Wylie et al. 2000]. To demonstrate that our protocol is efficient in practice, we compare its performance to BFT [Castro and Liskov 2001; 2002], the Byzantine fault-tolerant replicated state machine implementation that Castro and Liskov have made available [Castro and Rodrigues 2003]. Experiments show that PASIS scales better than BFT in terms of network utilization at the server and in terms of work performed by the server. Experiments also show that response times of PASIS and BFT are comparable.

3.1 System model

We describe the system infrastructure in terms of *clients* and *storage-nodes*. There are N storage-nodes and an arbitrary number of clients in the system.

A client or storage-node is *correct* in an execution if it satisfies its specification throughout the execution. A client or storage-node that deviates from its specification *fails*. Both clients and storage-nodes may suffer Byzantine faults. We make no assumptions about the behavior of Byzantine storage-nodes and Byzantine clients (e.g., we assume that Byzantine storage-nodes can collude with each other and with any Byzantine clients). We assume that Byzantine clients and storage-nodes are computationally bounded so that we can benefit from cryptographic primitives (i.e., cryptographic hash functions).

The protocol is developed with a hybrid storage-node failure model [Thambidurai and Park 1988]. Under a traditional hybrid failure model, up to t storage-nodes could fail, $b \leq t$ of which may be Byzantine faults; the remainder could only crash. However, we consider a hybrid failure model for storage-nodes that crash and recover. The crash-recovery failure model is a strict generalization of the omission and crash failure models.

First, we review the crash-recovery model from Aguilera et al. [Aguilera et al. 2000]. In a system of n processes, each process can be classified as always-up, eventually-up, eventually-down, or unstable. A process that is *always-up* never crashes. A process that is *eventually-up* crashes at least once, but there is a time after which it is permanently up. A process that is *eventually-down* crashes at least once, and there is a time after which it is permanently down. A process that is *unstable* crashes and recovers infinitely many times. These classifications are further refined: a process is *good* if it is either always-up or eventually-up.

We combine the crash-recovery model with the hybrid failure model as follows. Up to b storage-nodes may ever be Byzantine; such storage-nodes do not recover and are not *good*. There are at least $N - t$ good storage-nodes (where $b \leq t$). A storage-node that is not Byzantine is said to be *benign* (i.e., benign storage-nodes are either always-up, eventually-up, eventually-down, or unstable). We assume that storage-nodes have stable storage that persists throughout the crash and recovery process.

The protocol tolerates crash and Byzantine clients. As in any practical storage system, an authorized Byzantine client can write arbitrary values to storage. These writes only affect the value of the data, but do not compromise the safety (linearizability) of the object. A client that does not exhibit a Byzantine failure (it is either correct or crashes) is *benign*

We assume an asynchronous model of time (i.e., we make no assumptions about message transmission delays or the execution rates of clients and storage-nodes, except that it is non-zero). We assume point-to-point authenticated channels with properties similar to those used by Aguilera et al. [Aguilera et al. 2000]. In summary, channels do not create messages (*no creation*), channels may experience *finite duplication*, and channels are *fair loss*. The finite duplication property ensures that if benign process p sends a message to benign process q only a finite number of times, then q receives the message only a finite number of times. The fair loss property ensures that if benign process p sends infinitely many messages to good process q, then q receives infinitely many messages from p.

There are two types of *operations* in the protocol — *read operations* and *write operations* — both of which operate on *data-items*. Clients perform read/write operations that issue multiple read/write *requests* to storage-nodes. A read/write request operates on a *data-fragment*. A data-item is *encoded* into data-fragments. Clients may encode data-items in an erasure-tolerant manner; thus the distinction between data-item and data-fragment. Requests are *executed* by storage-nodes; a correct storage-node that executes a write request *hosts* that write operation.

Clients may encode data-items in an erasure-tolerant manner; thus the distinction between data-item and data-fragment. We focus here on threshold erasure codes in which any m of the n encoded data-fragments can decode the data-item. When $m = 1$, the replication is used. Examples of such codes are replication, Reed-Solomon codes [Berlekamp 1968], secret sharing [Shamir 1979], RAID 3/4/5/6 [Patterson et al. 1988], information dispersal (IDA) [Rabin 1989], short secret sharing [Krawczyk 1994], and “tornado” or LDPC codes [Luby et al. 2001].

Storage-nodes provide fine-grained versioning; correct storage-nodes host a version of the data-fragment for each write request they execute. There is a well known zero time, $\mathbf{0}$, and null value, $\perp$, which storage-nodes can return in response to read requests. Implicitly, all stored data is initialized to $\perp$ at time $\mathbf{0}$.

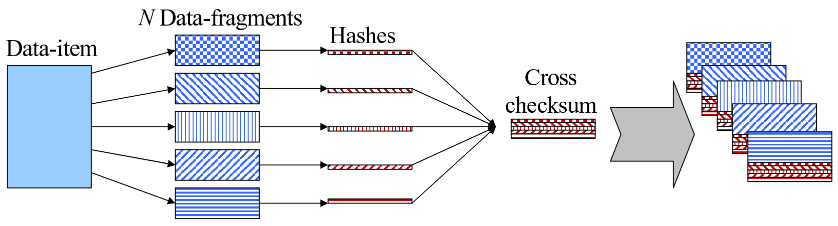


Figure 3.1: **Example of cross checksum generation for 5 data-fragments.** *To generate a cross checksum, a cryptographic hash is taken of each data-fragment. These hashes are then concatenated, replicated, and stored with each data-fragment.*

3.2 Mechanisms

This section describes mechanisms employed for encoding data, preventing Byzantine clients and storage-nodes from violating consistency, and authenticating client and storage-node requests. We assume that storage-nodes and clients are computationally bounded such that cryptographic primitives can be effective.

3.2.1 Erasure codes

We consider only threshold erasure codes in which any m of the N encoded data-fragments can decode the data-item; moreover, every m data-fragments can be used to deterministically generate the other $N - m$ data-fragments. We use a systematic information dispersal algorithm [Rabin 1989], which stripes the data-item across the first m data-fragments and generates erasure-coded data-fragments for the remainder. As such, each fragment is $\frac{1}{m}$ the total data size. This leads to a total size-blowup of $\frac{N}{m}$ for N fragments. Other threshold erasure codes (e.g., Secret Sharing [Shamir 1979] and Short Secret Sharing [Krawczyk 1994]) work as well.

3.2.2 Data-fragment integrity

Byzantine storage-nodes can corrupt their data-fragments. As such, it must be possible to detect and mask up to b storage-node integrity faults. Cross checksums [Gong 1989] enable read operations to detect corrupt data-fragments. A cryptographic hash of each data-fragment is computed, and the set of N hashes are concatenated to form the *cross*

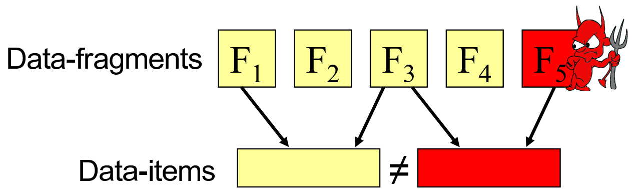


Figure 3.2: **Example of a "poisonous write" by a Byzantine client.** *In this example data-fragment 5 has been corrupted by a Byzantine client. Decoding different sets of fragments (i.e., 1 and 3 vs. 3 and 5) lead to data-item values that are not equivalent. Therefore, it is necessary to protect good clients from observing differing data values written to the same timestamp.*

checksum of the data-item. The cross checksum is stored with each data-fragment (i.e., it is replicated N times), enabling corrupted data-fragments to be detected at read time. Note, the N^2 space overhead is small relative to the data size, given reasonable data sizes (e.g., there is an 8.3% overhead for $N = 17$, $m = 5$, and a 16 KB block). An example of generating a cross checksum is shown in Figure 3.1.

3.2.3 Write operation integrity

Mechanisms are required to prevent Byzantine clients from performing a write operation that lacks integrity. If a Byzantine client generates arbitrary data-fragments (rather than erasure coding a data-item correctly), then each of the $\binom{N}{m}$ subsets of data-fragments could "recover" a distinct data-item. Additionally, a Byzantine client could partition the set of N data-fragments into subsets that each decode to a distinct data-item. These attacks are similar to *poisonous writes* for replication, as described by Martin et al. [Martin et al. 2002]. An example of a poisonous write is shown in Figure 3.2.

To protect against such Byzantine client actions, read operations must only return values that are written correctly (i.e., that are *single-valued*). To achieve this, the cross checksum mechanism is extended in two ways: validating timestamps combined with storage-node verification, and validated cross checksums.

Validating timestamps

To ensure that only a single set of data-fragments can be written at any logical time, the hash of the cross checksum is placed in the low order bits of the logical timestamp. Note, the hash is used for space-efficiency; instead, the entire cross checksum could be placed in the low bits of the timestamp.

On a write, each storage-node verifies its data-fragment against the corresponding hash in the cross checksum. The storage-node also verifies that the cross checksum matches the low-order bits of the validating timestamp. A correct storage-node only executes write requests for which both checks pass. Thus, a Byzantine client cannot make a correct storage-node appear Byzantine—only Byzantine storage-nodes can return unverifiable responses.

Validated cross checksums

Combining storage-node verification with validating timestamps ensures that the data-fragments being considered by any read operation were not fabricated by Byzantine storage-nodes. To ensure that the client that performed the write operation acted correctly, the cross checksum must be validated by the reader. For the reader to validate the cross checksum, all N data-fragments are required. Given any m data-fragments, the reader can generate the full set of N data-fragments a correct client should have written. The reader can then compute the "correct" cross checksum from the generated data-fragments. If the generated cross checksum does not match the validated cross checksum, then a Byzantine client performed the write operation. The example in Figure 3.3 shows the steps necessary to perform the validation described above.

Only a single-valued write operation can generate a cross checksum that can be validated. Instead of using validated cross checksums, our protocol could use Verifiable Secret Sharing [Chor et al. 1985; Feldman 1987]. Verifiable Secret Sharing enables storage-nodes to validate that the client acted correctly on each write request (instead of validating the data-item on each read operation).

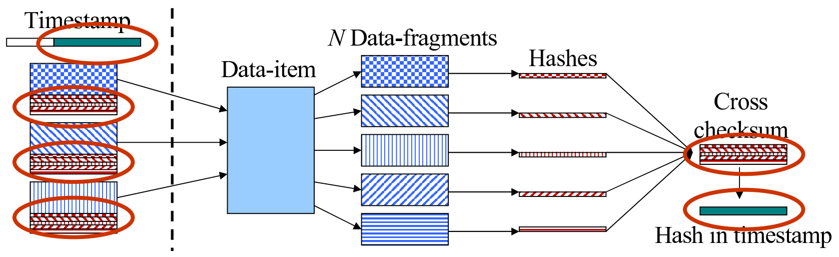


Figure 3.3: **Verification of a validating cross checksum at read time.** *This example shows the necessary steps to validate a set of 3 fragments at read time using a validated cross checksum. First, the full set of N data-fragments must be regenerated. Second, the cross checksum is computed and validated against the cross checksum that were read. Third, the hash of the cross checksum is taken and validated against the hash in the timestamp.*

3.2.4 Authentication

Clients and storage-nodes must be able to validate the authenticity of messages. RPC requests and responses are authenticated using HMACs (i.e., clients and storage-nodes have pair-wise shared secrets). Thus, the channels between clients and storage-nodes are authenticated. We assume some infrastructure is in place to distribute shared secrets—our implementation supports an existing Kerberos [Steiner et al. 1988] infrastructure.

3.3 Protocol

This section describes our Byzantine fault-tolerant consistency protocol that efficiently supports erasure-coded data-items by taking advantage of versioning storage-nodes. It describes, in detail, the protocol in pseudo-code form.

3.3.1 Overview

At a high level, the protocol proceeds as follows. Logical timestamps are used to totally order all write operations and to identify data-fragments pertaining to the same write operation across the set of storage-nodes. For each write, a logical timestamp is constructed by the client that is guaranteed to be unique (given the data and the logical time at which the data is being written) and greater than that of the *latest complete write* (the complete write with the highest timestamp). This is accomplished by querying storage-nodes for the greatest timestamp they host, and then incrementing the greatest response. In order to verify the integrity of the data, a hash performed over the data-fragment is appended to the logical timestamp. Two clients may arrive at the same logical timestamp only if they are both writing the same data concurrently to each other. In this case the write requests generated look identical to each other.

To perform a read operation, clients issue read requests to a subset of storage-nodes. Once at least a read quorum of storage-nodes reply, the client identifies the *candidate*—the response with the greatest logical timestamp. The set of read responses that share the timestamp of the candidate comprise the *candidate set*. The read operation *classifies* the candidate as *complete*, *repairable*, or *incomplete*. If the candidate is classified as complete, the data-fragments, timestamp, and return value are validated. If validation is successful, the value of the candidate is returned and the read operation is complete; otherwise, the candidate is reclassified as incomplete. If the candidate is classified as repairable, it is repaired by writing data-fragments back to the original set of storage-nodes (note, in [Malkhi and Reiter 1998b], repair, for replicas, is referred to as “write-back”). Prior to performing repair, data-fragments are validated in the same manner as for a complete candidate. If the candidate is classified as incomplete, the candidate is discarded, previous data-fragment versions are requested, and classification begins anew. Classification is performed according to a set of constraints dependent upon the failure model (see Section 3.4). All candidates fall into one of the three classifications, even those corresponding to concurrent or failed write operations.

3.3.2 Pseudo-code

The pseudo-code for the protocol is shown in Figures 3.5 and 3.6. The symbol LT denotes logical time and LT_{c} denotes the logical time of the candidate. The set $\{D_1, \ldots, D_N\}$ denotes the N data-fragments; likewise, $\{S_1, \ldots, S_N\}$ denotes the set of N storage-nodes. In the pseudo-code, the binary operator ‘$|$’ denotes string concatenation. Simplicity and clarity in the presentation of the pseudo-code was chosen over obvious optimizations that are in the actual implementation.

```
INITIALIZE() :
100: /* History tuples are ⟨LT, Data, CC⟩ */
101: /* History is stored in stable storage */
102: $History := \langle \mathbf{0}, \perp, \perp \rangle$

RECEIVE_READ_LATEST() :
200: /* Note, Latest is a singleton */
201: $Latest := (X : X.LT = \texttt{MAX}[History.LT], X \in History)$
202: SEND(READ_RESPONSE, $S$, $Latest$)

RECEIVE_WRITE_REQUEST($LT, D, CC$) :
300: if (VALIDATE_WRITE($LT, D, CC$)) then
301:     /* Execute the write request */
302:     $History := History \cup \langle LT, D, CC \rangle$
303:     SEND(WRITE_RESPONSE, $S$)
304: end if

VALIDATE_WRITE($LT, D, CC$) :
400: if ((HASH($CC$) $\neq$ $LT.Verifier$) OR (HASH($D$) $\neq$
       $CC[S]$)) then
401:     return (FALSE)
402: end if
403: /* Accept the write request */
404: return (TRUE)

RECEIVE_TIME_REQUEST() :
500: SEND(TIME_RESPONSE, $S$, MAX[$History.LT$])

RECEIVE_READ_PREVIOUS($LT$) :
600: $PreHistory := \{X : X.LT < LT, X \in History\}$
601: /* Note, Latest is a singleton */
602: $Latest := \texttt{MAX}[PreHistory.LT]$
603: SEND(READ_RESPONSE, $S$, $Latest$)
```

Figure 3.4: **Pseudo-code for storage-node S.**

Storage-node interface

Storage-nodes offer interfaces to write a data-fragment at a specific logical time; to query the greatest logical time of a hosted data-fragment; to read the hosted data-fragment with the greatest logical time; and to read the hosted data-fragment with the greatest logical time at or before some logical time. Each write request a storage-node executes creates a new version of the data-fragment (indexed by its logical timestamp) at the storage-node (i.e., the storage-node performs comprehensive versioning).

All stored data is initialized to $\perp$ at time $\mathbf{0}$, and has a cross checksum of $\perp$. The zero time, $\mathbf{0}$, and the null value, $\perp$, are well known values which the clients understand. The storage-node pseudo-code is shown in Figure 3.4. The *History* which contains the version history for the data-item is kept in stable storage such that it persists during a crash and subsequent recovery. Storage-nodes validate write requests before executing them (to protect against Byzantine clients). This is performed by the function VALIDATE_WRITE called by RECEIVE_WRITE_REQUEST. The value returned by RECEIVE_READ_LATEST and RECEIVE_READ_PREVIOUS, *Latest*, is guaranteed to be unique, since timestamps are unique (i.e., two distinct write operations cannot have the same timestamp).

```
WRITE($Data$) :
100: /* Encode data, construct timestamp, and write data-
       fragments */
101: $\{D_1, \ldots, D_N\} :=$ ENCODE($Data$)
102: $CC :=$ MAKE_CROSS_CHECKSUM($\{D_1, \ldots, D_N\}$)
103: $Time :=$ READ_TIMESTAMP()
104: $LT :=$ MAKE_TIMESTAMP($Time, CC$)
105: DO_WRITE($\{D_1, \ldots, D_N\}$, $LT$, $CC$)

READ_TIMESTAMP() :
200: $ResponseSet := \emptyset$
201: repeat
202:   for all $S_i \in \{S_1, \ldots, S_N\} \setminus ResponseSet.S$ do
203:     SEND($S_i$, TIME_REQUEST)
204:   end for
205:   if (POLL_FOR_RESPONSE() = TRUE) then
206:     $\langle S, LT \rangle :=$ RECEIVE_TIME_RESPONSE()
207:     if ($S \notin ResponseSet.S$) then
208:       $ResponseSet := ResponseSet \cup \langle S, LT \rangle$
209:     end if
210:   end if
211: until ($|ResponseSet| = N - t$)
212: return (MAX[$ResponseSet.LT.Time$] $+ 1$)

MAKE_CROSS_CHECKSUM($\{D_1, \ldots, D_N\}$) :
300: for all $D_i \in \{D_1, \ldots, D_N\}$ do
301:   $H_i :=$ HASH($D_i$)
302: end for
303: $CC := H_1 | \ldots | H_N$
304: return ($CC$)

MAKE_TIMESTAMP($Time, CC$) :
400: $LT.Time := Time$
401: $LT.Verifier :=$ HASH($CC$)
402: return ($LT$)

DO_WRITE($\{D_1, \ldots, D_N\}$, $LT$, $CC$) :
500: $ResponseSet := \emptyset$
501: repeat
502:   for all $S_i \in \{S_1, \ldots, S_N\} \setminus ResponseSet.S$ do
503:     SEND($S_i$, WRITE_REQUEST, $LT$, $D_i$, $CC$)
504:   end for
505:   if (POLL_FOR_RESPONSE() = TRUE) then
506:     $\langle S \rangle :=$ RECEIVE_WRITE_RESPONSE()
507:     if ($S \notin ResponseSet.S$) then
508:       $ResponseSet := ResponseSet \cup \langle S \rangle$
509:     end if
510:   end if
511: until ($|ResponseSet| = N - t$)
```

Figure 3.5: **Client-side write operation pseudo-code.**

Write operation

The WRITE operation, shown in Figure 3.5 consists of determining the greatest logical timestamp, constructing write requests, and issuing the requests to the storage-nodes. First, a timestamp greater than, or equal to, that of the latest complete write must be determined. Collecting $N-t$ responses, on line 211 of READ_TIMESTAMP, ensures that the response set intersects a complete write at a correct storage-node. Since the environment is asynchronous, a client can wait for no more than $N-t$ responses. Fewer than $N-t$ responses are actually required to observe the timestamp of the latest complete write, since a single correct response is sufficient; in fact, this bound is $t+b+1$.

Next, the ENCODE function, on line 101 of WRITE, encodes the data-item into N data-fragments. The data-fragments are used to construct a cross checksum from the concatenation of the hash of each data-fragment (line 102). The function MAKE_TIMESTAMP, called on line 104, generates a logical timestamp to be used for the current write operation. This is done by incrementing the high order bits of the greatest observed logical timestamp from the *ResponseSet* (i.e., $LT.TIME$) and appending the *Verifier*. The *Verifier* is just the hash of the cross checksum.

Finally, write requests are issued to all storage-nodes. Each storage-node is sent a specific data-fragment, the logical timestamp, and the cross checksum. A storage-node validates the cross checksum with the verifier and validates the data-fragment with the cross checksum before executing a write request (i.e., storage-nodes call `VALIDATE_WRITE` listed in their pseudo-code). The write operation returns to the issuing client once $N-t$ `WRITE_RESPONSE` messages are received (line 511 of `DO_WRITE`).

Read operation

The read operation, shown in Figure 3.6, iteratively identifies and classifies candidates, until a repairable or complete candidate is found. Once a repairable or complete candidate is found, the read operation validates its correctness and returns the data. Note that the read operation returns a $\langle timestamp, value\rangle$ pair; in practice, a client only makes use of the value returned.

The read operation begins by issuing `READ_LATEST` commands to all storage-nodes (via the `DO_READ` function). Each storage-node responds with the data-fragment, logical timestamp, and cross checksum corresponding to the greatest timestamp it has executed.

The integrity of each response is individually validated through the `VALIDATE` function, called on line 207 of `DO_READ`. This function checks the cross checksum against the *Verifier* found in the logical timestamp and the data-fragment against the appropriate hash in the cross checksum.

A second type of validation is performed on read responses (also on line 207). For responses to `READ_PREV` commands, the logical timestamp is checked to ensure it is strictly less than the timestamp specified in the command. This check ensures that improper responses from Byzantine storage-nodes are not included in the response set.

Since, in an asynchronous system, slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N-t$ read responses can be collected (line 211 of `DO_READ`).

```
READ() :
100: $ResponseSet := \texttt{DO\_READ}(\texttt{READ\_LATEST}, \perp)$
101: loop
102:     $\langle CandidateSet, LT_c \rangle :=$
             $\texttt{SELECT\_CS}(ResponseSet)$
103:     if $(|CandidateSet| \geq \texttt{INCOMPLETE}$ then
104:         /* Complete or repairable write found */
105:         $\{D_1, \ldots, D_N\} := \texttt{GEN\_FRAGS}(CandidateSet)$
106:         $CC := \texttt{MAKE\_CROSS\_CHECKSUM}(\{D_1, \ldots, D_N\})$
107:         if $(CC = CandidateSet.CC)$ then
108:             /* Cross checksum is validated */
109:             if $(|CandidateSet| < \texttt{COMPLETE})$ then
110:                 /* Repair is necessary */
111:                 $\texttt{DO\_WRITE}(\{D_1, \ldots, D_N\}, LT_c, CC)$
112:             end if
113:             $Data := \texttt{DECODE}(\{D_1, \ldots, D_N\})$
114:             return $(\langle LT_c, Data \rangle)$
115:         end if
116:     end if
117:     /* Incomplete or validation failed, loop again */
118:     $ResponseSet := \texttt{DO\_READ}(\texttt{READ\_PREV}, LT_c)$
119: end loop

DO_READ(READ_COMMAND, LT) :
200: $ResponseSet := \emptyset$
201: repeat
202:     for all $S_i \in \{S_1, \ldots, S_N\} \setminus ResponseSet.S$ do
203:         $\texttt{SEND}(S_i, \texttt{READ\_COMMAND}, LT)$
204:     end for
205:     if $(\texttt{POLL\_FOR\_RESPONSE}() = \texttt{TRUE})$ then
206:         $\langle S, Resp \rangle := \texttt{RECEIVE\_READ\_RESPONSE}()$
207:         if $((\texttt{READ\_COMMAND} = \texttt{READ\_LATEST}$ OR
                 $Resp.LT < LT)$ AND
                 $(S \notin ResponseSet.S)$ AND
                 $(\texttt{VALIDATE}(Resp.D, Resp.CC, Resp.LT, S)))$
           then
208:             $ResponseSet := ResponseSet \cup \langle S, Resp \rangle$
209:         end if
210:     end if
211: until $(|ResponseSet| = N - t)$
212: return $(ResponseSet)$

VALIDATE(D, CC, LT, S) :
300: if $((\texttt{HASH}(CC) \neq LT.Verifier)$ OR $(\texttt{HASH}(D) \neq$
       $CC[S]))$ then
301:     return (FALSE)
302: end if
303: return (TRUE)
```

Figure 3.6: **Client-side read operation pseudo-code.**

Since correct storage-nodes perform the same validation before executing write requests, the only responses that can fail the client's validation are those from Byzantine storage-nodes. For every discarded Byzantine storage-node response, an additional response can be awaited.

After sufficient responses have been received, a candidate for classification is chosen. The function SELECT_CS, called on line 102 of READ, determines the candidate timestamp, denoted LT_c, which is the greatest timestamp found in the response set. All data-fragments that share LT_c are identified and returned as the candidate set. At this point, the candidate set contains a set of validated data-fragments that share a common cross checksum and logical timestamp.

Once a candidate has been chosen, it is classified as either complete, repairable, or incomplete based on the size of the *CandidateSet*. The rules for classifying a candidate as INCOMPLETE or COMPLETE are given in the following subsection. If the candidate is classified as incomplete, a READ_PREV message is sent to each storage-node with its timestamp. Candidate classification begins again with the new response set.

If the candidate is classified as either complete or repairable, the candidate set contains sufficient data-fragments written by the client to decode the original data-item. To validate the observed write's integrity, the candidate set is used to generate a new set of data-fragments (line 105 of READ). A validated cross checksum, CC_{valid}, is computed from the newly generated data-fragments. The validated cross checksum is compared to the cross checksum of the candidate set (line 107 of READ). If the check fails, the candidate was written by a Byzantine client; the candidate is reclassified as incomplete and the read operation continues. If the check succeeds, the candidate was written by a correct client and the read enters its final phase. Note that this check either succeeds or fails for all correct clients regardless of which storage-nodes are represented within the candidate set.

If necessary, repair is performed: write requests are issued with the generated data-fragments, the validated cross checksum, and the logical timestamp (line 109 of READ). Storage-nodes not currently hosting the write execute the write at the given logical time; those already hosting the write are safe to ignore it. Finally, the function DECODE, on line 113 of READ, decodes m data-fragments, returning the data-item.

It should be noted that, even after a write completes, it may be classified as repairable by a subsequent read, but it will never be classified as incomplete. For example, this could occur if the read set (of $N - t$ storage-nodes) does not fully encompass the write set (of $N - t$ storage-nodes).

3.4 Constraints

The symbol Q_C denotes a complete write operation: the number of benign storage-nodes that must execute write responses for a write operation to be complete. Note that since threshold quorums are used, Q_C is a scalar value. To ensure that linearizability and liveness are achieved, Q_C and N must be constrained with regard to b, t, and each other. As well, the parameter m, used in DECODE, must be constrained. We sketch safety and liveness proofs for the protocol in Appendix A.

3.4.1 Read classification

To classify a candidate as `COMPLETE`, a candidate set of at least Q_C benign storage-nodes must be observed. In the worst case, at most b members of the candidate set may be Byzantine, thus,

$$|CandidateSet| - b \geq Q_C \Rightarrow \texttt{COMPLETE}. \tag{3.1}$$

To classify a candidate as `INCOMPLETE` a client must determine that a complete write does not exist in the system (i.e., fewer than Q_C benign storage-nodes host the write). For this to be the case, the client must have queried all possible storage-nodes $(N-t)$, and must assume that nodes not queried host the candidate in consideration. So,

$$|CandidateSet| + t < Q_C \Rightarrow \texttt{INCOMPLETE}. \tag{3.2}$$

3.4.2 Real repairable candidates

To ensure that Byzantine storage-nodes cannot fabricate a repairable candidate, a candidate set of size b must be classifiable as incomplete. Substituting b into (3.2),

$$b + t < Q_C. \tag{3.3}$$

3.4.3 Decodable repairable candidates

Any repairable candidate must be decodable. The lower bound on candidate sets that are repairable follows from (3.2) (since the upper bound on classifying a candidate as incomplete coincides with the lower bound on repairable):

$$1 \leq m \leq Q_C - t. \tag{3.4}$$

Protocol	**Asynchronous repairable**
N	$2t+2b+1 \leq N$
Q_C	$t+b+1 \leq Q_C \leq N-t-b$
m	$1 \leq m \leq Q_C - t$
Complete	$\lvert CandidateSet \rvert \geq Q_C + b$
Incomplete	$\lvert CandidateSet \rvert < Q_C - t$

Table 3.1: **Protocol constraint summary**

3.4.4 Write termination

To ensure write operations are able to complete in an asynchronous environment in the face of b Byzantine storage-nodes,

$$Q_C + b \leq N - t, Q_C \qquad \leq N - t - b. \tag{3.5}$$

Since slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N-t$ responses can be awaited. As well, b responses received may be from Byzantine storage-nodes.

3.4.5 Constraint summary

The summary of constraints is given in Table 3.1. The bounds on N (i.e., $N > 2t + 2b$) have been shown to be optimal for systems with single round-trip write operations [Abraham et al. 2004]. A diagram that more intuitively shows the constraint on N is shown in Figure 3.7.

3.5 Implementation

PASIS consists of clients and storage-nodes. Storage-nodes store data-fragments and their versions. Clients execute the protocol to read and write data-items.

3.5.1 Storage-node implementation

Storage-nodes use the Comprehensive Versioning File System (CVFS) [Soules et al. 2003] to retain data-fragments and their versions. CVFS uses a log-structured data or-

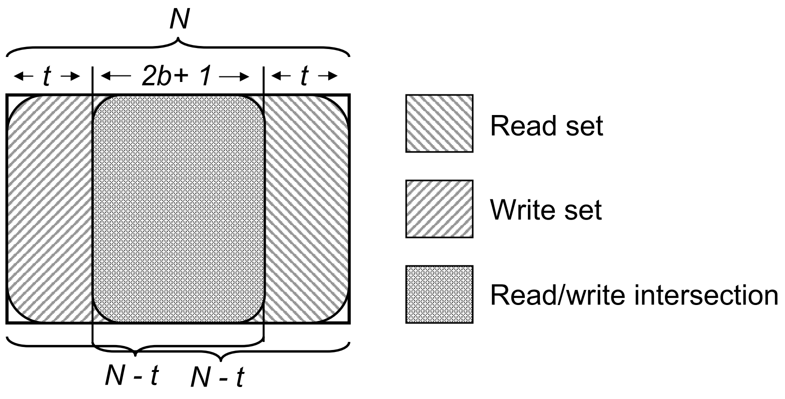


Figure 3.7: **Illustration of constraint on N.** *This figure shows the intuitive reasoning behind the constraint on $N = 2t+2b+1$. As shown, a write executes at any $N-t$ storage-nodes, and a read executes at a set of $N-t$ storage-nodes that has the minimum overlap with the write. The read only observes the data value on the storage-nodes within the intersection of the read and write. Since b of these storage-nodes may be Byzantine $b+1$ matching values must be observed. This leads to an intersection of size $2b+1$. Thus, $t+(2b+1)+t = N = 2t+2b+1$.*

ganization to reduce the cost of data versioning. Experience indicates that retaining every version and performing local garbage collection comes with minimal performance cost (a few percent) and that it is feasible to retain complete version histories for several days [Soules et al. 2003; Strunk et al. 2000].

We extended CVFS to provide an interface for retrieving the logical timestamp of a data-fragment. Implicitly, each write request creates a new version of the data-fragment (indexed by its logical timestamp) at the storage-node. In addition to data, each write request contains a cross checksum, a logical timestamp, and a *linkage record* [Amiri et al. 1999]. The linkage record consists of descriptions of the encoding scheme, and addresses of the N storage-nodes for a specific data-item; it is fixed upon data-item creation.

By default, a read request returns the most current data-fragment version, ordered by logical timestamp. Read responses may also contain a limited version history containing logical timestamps of previously executed write requests. The version history allows clients to identify and classify additional candidates without issuing extra read requests. Storage-node can also return read responses that contain no data other than version histories, which makes candidate classification more network-efficient.

3.5.2 Garbage collection

Pruning old versions, or garbage collection (GC), is necessary to prevent capacity exhaustion of the storage-nodes. A storage-node in isolation cannot determine which local data-fragment versions are safe to garbage-collect, because write completeness is a property of a set of storage-nodes. A data-fragment version can be garbage-collected only if there exists a later complete write for the corresponding data-item. Storage-nodes can classify writes by executing the read protocol in the same manner as a client. However, no data need be returned for protocol members that do not tolerate Byzantine clients (since the cross checksum need not be validated). Linkage records provide sufficient information for the storage-nodes to know which other nodes host relevant data-fragments.

Garbage collection is implemented in the current prototype and it requires no additional RPCs. We have implemented a heuristic to invoke GC whenever idle time is detected. Periodically, a thread wakes up and checks the system load. If the system load is low, then GC will be invoked. We use a timer-based idle-time detector, as described by Golding et al.[Golding et al. 1995]. This type of idle time detector was successfully used in cleaning heuristics for LFS, even in heavily loaded systems [Blackwell et al. 1995]. We also have a method for invoking GC externally of the system (e.g., by a CRON job).

It is usually inefficient to perform GC for every block, since most blocks do not have old versions that need to be reclaimed (e.g., in a system with a read-heavy workload). We address this by adding a counter to the PASIS per-block metadata to track the number of writes to a block. This counter is incremented during each write operation and reset after GC has run. If the the block's write-count rises above certain threshold, an entry identifying the block is added to an in-memory *high-write-count* table. When GC is executed, it first searches this table. If an entry is found, it is removed and GC is executed on that block. If no entries are found, GC can scan the block-space sequentially. Although this heuristic works, further research into policy issues, such as the appropriate frequency and order of garbage collection, is warranted.

3.5.3 Client implementation

Our client implementation follows the pseudo-code described above. The client module is accessed through a set of library interface calls. These calls allow an application to control the encoding scheme, the threshold values, and the failure and timing models. The client protocol routines are implemented such that different protocol family members and thresholds may be specified for different data-items. Likewise, the storage-nodes for any given data-item are also specified via these interfaces, thus externalizing control (and responsibility) for such bootstrapping information; for our experiments we use a static set of N storage-nodes. Clients communicate with storage-nodes through a TCP-based RPC interface.

In an asynchronous environment, the client implementation issues `TIME_REQUEST` requests to only $N+b-Q_C$ storage-nodes, since this ensures overlap with the latest complete write. To improve the responsiveness of write operations, clients return after the first Q_C+b storage-nodes respond; the remainder of the requests complete in the background.

To improve the read operation's performance, only m read requests fetch the latest data pertaining to the data-fragment, while all receive version histories; this makes the read operation more network-efficient. The limited data-fragment version history returned by read requests, allows clients to classify earlier writes without issuing additional storage-node requests. If necessary, after classification, extra data-fragments are fetched according to the candidate's timestamp. Once the data-item is successfully validated and decoded, it is returned.

3.5.4 Erasure codes

In our erasure coding implementation, if $m=1$, then replication is employed, otherwise an information dispersal algorithm [Rabin 1989] is used. Our information dispersal implementation stripes the data-item across the first m data-fragments (i.e., each data-fragment is $\frac{1}{m}$ of the original data-item's size). This makes the erasure code a *systematic encoding*; Thus, concatenation of the first m data-fragments produce the original data-item. These *stripe-fragments* are used to generate the *code-fragments* via polynomial interpolation within a Galois Field, which treats the stripe-fragments and code-fragments as points on some $m-1$ degree polynomial. Our implementation of polynomial interpolation was originally based on [Dai 2003] (which conceptually follows [Rabin 1989]). We modified the source to use stripe-fragments and added an implementation of Galois Fields of size 2^8 that use lookup tables for multiplication.

Beyond our base erasure code implementation, we implemented secret sharing [Shamir 1979] and short secret sharing [Krawczyk 1994]. Our implementation of short secret sharing closely follows [Krawczyk 1994], using AES for the cipher. Such erasure codes can also provide a degree of confidentiality with regard to storage-nodes.

Our implementation of cross checksums closely follows Gong [Gong 1989]. Our implementation uses a publicly available implementation of MD5 [Rivest 1992] for all hashes. Each MD5 hash is 16 bytes long; thus, each cross checksum is $N \times 16$ bytes long.

3.6 Evaluation

This section evaluates protocol family performance in the context of the prototype block storage system.

3.6.1 Experimental setup

We use a cluster of 20 machines to perform experiments. Each storage-node is a dual 1GHz Pentium III machine with 384 MB of memory and a 9GB Quantum Atlas 10K disk. Each client is a single processor 2GHz Pentium IV machine. The machines are connected through a 100Mb switch. All machines run the Linux 2.4.20 SMP kernel.

In all experiments, clients keep a fixed number of read and write operations outstanding; when an operation completes, a new operation is issued immediately. Unless otherwise specified, requests are for random 16 KB blocks. Unless otherwise specified, Q_C and N are the minimum allowable values for the protocol member, as given in Table 3.1, and m is the maximum allowable value. Authentication costs (i.e., HMAC computations) are included in all experiments.

3.6.2 Performance and scalability of PASIS protocol members

PASIS configuration

Each storage-node is configured with 128 MB of data cache, and no caching is done on the clients. Storage-nodes use write-back caching, mimicking availability of 16 MB of non-volatile RAM. All experiments focus on the protocol costs: the working sets fit into memory and all caches are warmed up beforehand. Results from such experiments highlight the overheads introduced by the protocol and not those introduced by the disk system. It is, however, a full system implementation: each storage-node is backed by a real persistent data store, and compulsory cache flushes are serviced by the disk system.

Space-efficiency of protocol members

All protocol members can employ m-of-n erasure codes. Increasing m improves space-efficiency, since each data-fragment is $\frac{1}{m}$ the size of the data-item. Space-efficiency reduces the network bandwidth needed, which reduces the response time of operations.

To perform a write operation, N data-fragments are sent over the network. With each data-fragment, a cross checksum and linkage record are sent. Respectively, these are N times the size of a MD5 digest (16 bytes) and N times the size of a storage-node ID (4 bytes). Thus, the network bandwidth consumed by cross checksums is $20 \cdot N^2$ bytes. RPC headers and arguments consume negligible bandwidth. Thus, the total amount of data sent over the network by a write operation is: $16\text{ KB} \times \frac{N}{m} + 20\text{ B} \times N^2$.

Computation costs

Computation costs are incurred to erasure code data. Additional computation costs are incurred to authenticate messages and protect against non-crash failures. The majority of such computation costs are paid by clients in the system, rather than storage-nodes.

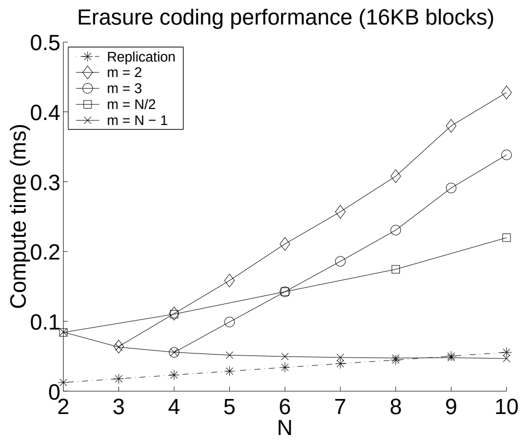


Figure 3.8: **Computational cost of erasure coding.** *Block size, N, and m dictate the computational cost of erasure coding.*

Erasure coding costs. Figure 3.8 shows the trends in the cost of encoding data with our erasure code implementation. For comparison, the performance of N-fold replication (i.e., N `memcpys`) is shown. Lines are shown for fixed m values of two and three. These lines illustrate that, as expected, the cost of an erasure code for a given m grows linearly with N, since the number of code-fragments grows with N.

Two other lines are shown in Figure 3.8 to illustrate the interesting impact of m on performance: the space-efficiency of an erasure code is inversely proportional to m whereas the cost of generating some aggregate amount of code-fragment is proportional to m. Consider the $m = \frac{N}{2}$ line. For each point on the line, erasure coding generates, in total, 16 KB of code-fragments, although the number and individual sizes of the code-fragments differ. When generating some aggregate amount of code-fragments, the cost of erasure coding grows linearly with m. For $m = N - 1$, a single code-fragment is needed for each write; as expected, the cost of generating one fragment decreases with N, since the size of the fragment also decreases (to $\frac{1}{N-1}$).

COMPUTATION COST BREAKDOWN: Table 3.2 enumerates the client and storage-node computation costs for the protocol tolerating one and four Byzantine storage-node faults (i.e., $b = t = 1$ and for $b = t = 4$).

CLIENTS: All protocol family members place the majority of the computational work on

	$b = t = 1$	$b = t = 4$
Storage-node: write operation costs		
Verify timestamp	1.56 μs	3.78 μs
Verify data-fragment	72.2	29.4
Client: write operation costs		
Encode: generate $N - m$ code-fragments	163	546
Generate one code-fragment	*54.2*	*45.5*
Generate cross checksum: hash N data-fragments	359	512
Hash one data-fragment	*71.2*	*30.1*
Generate validating timestamp	1.60	3.72
Client: read operation costs		
Verify data-fragments: hash m data-fragments	143	150
Best case decode: `memcpy` m stripe-fragments	6.84	7.86
Worst case decode: generate m code-fragments	108	228
Validate cross checksum (to tolerate Byz. clients)	522	1060

Table 3.2: **Client and storage-node computation costs.** *Costs are broken down for the asynchronous repairable protocol member with Byzantine storage-nodes for: $b = t = 1$ and $b = t = 4$ ($N = 5,\ m = 2$ and $N = 17,\ m = 5$, respectively).*

clients in the system. Erasure-coding is done by the client and requires nothing of the storage-node. The difference in computation costs for the two instances of the protocol member listed is due to their respective values of N and m. The cost of erasure coding with regard to N and m is discussed above. The cost of generating cross checksums grows with $\frac{N}{m}$.

Read operations in protocol members with only crash clients are computationally less demanding than write operations. A read operation requires fewer hashes of data-fragments and generation of fewer code-fragments. In the best case, the m stripe-fragments can be concatenated and no code-fragments need be generated. In protocol members that tolerate Byzantine clients, read operations performs almost the same computation as write operations to validate the cross checksum (i.e., $N - m$ code-fragments are generated and N data-fragments hashes are taken).

Short secret sharing can be used in place of our default erasure code. Doing so adds

$\approx$550 μs to the base erasure code costs for encrypting the data-item under the AES cipher and less than 20 μs for generating and secret sharing the encryption key (this cost depends on m and N). Both write and read operations incur these costs.

STORAGE-NODES: For each write request, a storage-node must verify both the timestamp and the data-fragment. Validating the data-fragment is roughly $\frac{1}{N}$ the work the client does in creating the cross checksum. A hash of the cross checksum is taken to verify the hash within the timestamp. Read requests require no significant computation by the storage-node (for the protocol).

AUTHENTICATION: Clients and storage-nodes must authenticate each RPC request and response. Authentication is performed over the RPC header and some RPC arguments. Cross checksums and data-fragments are not directly included in the authentication; however, the validating timestamp is included, and it indirectly authenticates the remainder. In all cases, authentication of an RPC message requires less than 2.5 μs.

3.6.3 Performance and scalability comparison with BFT

BFT configuration

We compare the PASIS implementation of our protocol with the BFT library implementation [Castro and Rodrigues 2003] of the BFT protocol for replicated state machines [Castro and Liskov 1998a], since it is generally regarded as efficient. The semantics provided by BFT are stronger than those provided by the PASIS read/write protocol. Since BFT implements a Byzantine fault-tolerant replicated state machine, arbitrary operations are linearizable, not just block reads and writes (as in PASIS). Additionally, the bounds on the number of storage-nodes required to tolerate an equivalent number of faults are lower than in PASIS ($3b+1$ for BFT as compared with $4b+1$ in PASIS). However, BFT incurs the cost of multiple rounds of server communication.

Operations in BFT require agreement among the replicas (storage-nodes in PASIS). Agreement is performed in four steps: (i) the client broadcasts requests to all replicas; (ii) the *primary* broadcasts pre-prepare messages to all replicas; (iii) all replicas broadcast prepare messages to all replicas; and, (iv) all replicas send replies back to the client and then broadcast commit messages to all other replicas. Commit messages are piggy-backed on the next pre-prepare or prepare message to reduce the number of messages on the network. *Authenticators*, lists of MACs, are used to ensure that broadcast messages from clients and replicas cannot be modified by a Byzantine replica. All clients and replicas have public and private keys that enables them to exchange symmetric cryptography keys used to create MACs. Logs of commit messages are checkpointed (garbage collected) periodically. View changes, in which a new primary is selected, are suppressed in all experiments.

An optimistic fast path for read operations (i.e., operations that do not modify state) is implemented in BFT. The client broadcasts its request to all replicas. Each replica replies once all messages previous to the request are committed. Only one replica sends the full reply (i.e., the data and digest), and the remainder just send digests that can verify the correctness of the data returned. If the replies from replicas do not agree, the client re-issues the read operation—for the replies to agree, the read-only request must arrive at $2b+1$ of the replicas in the same order (with regard to other write operations). Re-issued read operations perform agreement using the base BFT algorithm.

The BFT configuration does not store data to disk: instead, it stores all data in memory and accesses it via memory offsets (i.e., we implemented a simple block interface using BFT). As such, the storage-component of BFT is much faster than PASIS' storage component, which provides true disk-based storage. The difference in storage-component latency can be seen in the BFT vs. PASIS breakdown graph shown in Figure 3.10.

BFT uses UDP connections rather than TCP. BFT's retransmission policy is static (i.e., it does not adapt with the detection of congestion as does TCP's policy) and can only be set at a coarse granularity (milliseconds). We have observed high rates of retransmission at high load when running write workloads using BFT on our LAN. We believe this causes the dropoff in write throughput as shown in Figure 3.11. Due to the BFT library's code structure, it would require significant work to change the transport to use TCP instead of UDP in order to achieve a fairer comparison.

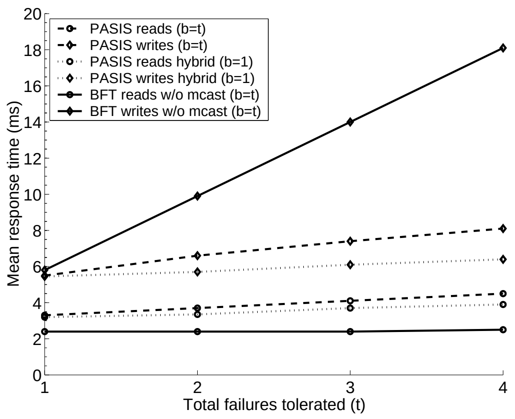


Figure 3.9: **Mean response time vs. total failures tolerated.** *Mean response times of read and write operations of random 16 KB blocks in PASIS and BFT. Lines are shown for PASIS that correspond to both $b = t$ and $b = 1$ (a hybrid fault model). Multicast was not used for these BFT experiments.*

The BFT implementation defaults to using IP multicast. In our environment, like many, IP multicast broadcasts to the entire subnet, thus making it unsuitable for shared environments. We found that the BFT implementation code is fairly fragile when using IP multicast in our environment, making it necessary to disable IP multicast in some cases (where stated explicitly). The BFT implementation authenticates broadcast messages via authenticators, and point-to-point messages with MACs.

Response time

Figure 3.9 shows the mean response time of a single request from a single client as a function of tolerated number of storage-node failures. Due to the fragility of the BFT implementation with $b > 1$, IP multicast was disabled for BFT during this experiment. The focus in this plot is the slopes of the response time lines: the flatter the line the more scalable the protocol is with regard to the number of faults tolerated. In our environment, a key contributor to response time is network cost, which is dictated by the space-efficiency of the protocol.

Figure 3.10 breaks the mean response times of read and write operations, from Figure 3.9, into the costs at the client, on the network, and at the storage-node for $b = 1$ and

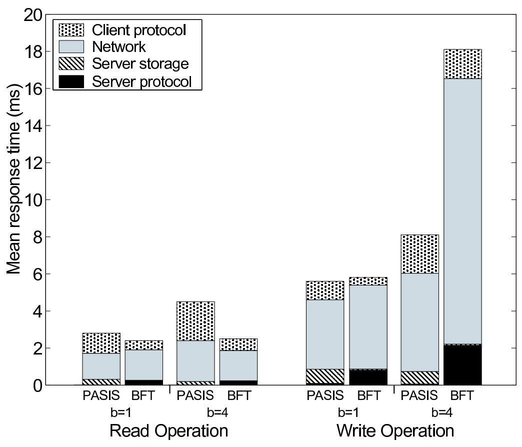


Figure 3.10: **Protocol cost breakdown.** *The bars illustrate the cost breakdown of read and write operations for PASIS and BFT for* $b = 1$ *and* $b = 4$*. Each bar corresponds to a single point on the mean response time graph in Figure 3.9. BFT does not store data to disk, as such no server storage cost is shown for BFT.*

$b = 4$. Since measurements are taken at the user-level, kernel-level timings for host network protocol processing (including network system calls) are attributed to the “network” cost of the breakdowns. To understand the response time measurements and scalability of these protocols, it is important to understand these breakdowns.

PASIS has better response times than BFT for write operations due to the space-efficiency of erasure codes and the nominal amount of work storage-nodes perform to execute write requests. For $b = 4$, BFT has a blowup of $13\times$ on the network (due to replication), whereas our protocol has a blowup of $\frac{17}{5} = 3.4\times$ on the network. With IP multicast the response time of the BFT write operation would improve significantly, since the client would not need to serialize 13 replicas over its link. However, IP multicast does not reduce the aggregate server network utilization of BFT—for $b = 4$, 13 replicas must be delivered.

PASIS has longer response times than BFT for read operations. This can be attributed to two main factors: First, the PASIS storage-nodes store data in a real file system; since the BFT-based block store keeps all data in memory and accesses blocks via memory offsets, it incurs almost no server storage costs. We expect that a BFT implementation with actual data storage would incur server storage costs similar to PASIS (e.g., around

0.7 ms for a write and 0.4 ms for a read operation, as is shown for PASIS with $b = 1$ in Figure 3.10). Indeed, the difference in read response time between PASIS and BFT at $b = 1$ is mostly accounted for by server storage costs. Second, for our protocol, the client computation cost grows as the number of failures tolerated increases because the cost of generating data-fragments grows as N increases.

In addition to the $b = t$ case, Figure 3.9 shows one instance of PASIS assuming a hybrid fault model with $b = 1$. For space-efficiency, we set $m = t + 1$. Consequently, $Q_C = 2t + 1$ and $N = 3t + 2$. At $t = 1$, this configuration is identical to the Byzantine-only configuration. As t increases, this configuration is more space-efficient than the Byzantine-only configuration, since it requires $t - 1$ fewer storage-nodes. As such, the response times of read and write operations scale better.

Some read operations in PASIS can require repair. A repair operation must perform a "write" operation to repair the value before it is returned by the read. Interestingly, the response time of a read that performs repair is less than the sum of the response times of a normal read and a write operation. This is because the "write" operation during repair does not need to read logical timestamps before issuing write requests. Additionally, data-fragments need only be written to storage-nodes that do not already host the write operation.

Throughput

Figure 3.11 shows the throughput in 16 KB requests per second as a function of the number of clients (one request per client) for $b = 1$. Read and write operations are evaluated separately. Since $b = 1$ in this experiment, BFT uses multicast (which greatly improves its network efficiency). PASIS was run in two configurations: one with the thresholds set to that of the minimum system with $m = 2$, $N = 5$ (write blowup of $2.5\times$) and one, more space-efficient, with $m = 3$, $N = 6$ (write blowup of $2\times$). For these experiments, the data working set fit within the PASIS storage-node caches. Results indicate that, at high client load, throughput is limited by the server network bandwidth. If the working set were to exceed the cache size, PASIS would experience capacity misses that would incur disk

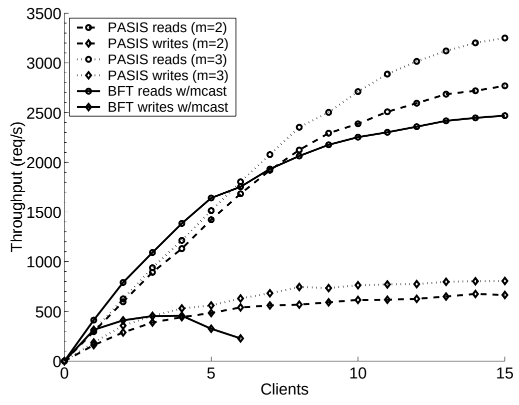


Figure 3.11: **Throughput vs. number of clients (**$b = 1$**).** *Throughput of read and write operations of random 16 KB blocks in PASIS and BFT for b* $= 1$. *Each client had one request outstanding. For PASIS, lines corresponding to both m* $= 2, N = 4$ *and m* $= 3, N = 5$ *are shown. For BFT, multicast was used.*

accesses. At this point, the disk subsystem would become the bottleneck.

At high load, PASIS has greater write throughput than BFT. BFT's write throughput peaks at 456 requests per second. But, we observed BFT's write throughput drops off significantly as client load increased; during these drop-offs, we observed a large increase in request retransmissions. We believe that this is due to the use of UDP and a coarse-grained retransmit policy in BFT's implementation. The write throughput of PASIS begins to flatten out at 675 requests per second for $m = 2$ and 806 req/sec for $m = 3$, significantly outperforming BFT. PASIS provides higher write throughput than BFT, because server links become bottlenecks, even though multicast is used.

Even with multicast enabled, each BFT server link sees a full 16 KB replica, whereas each PASIS server link sees $\frac{16}{m}$ KB. Similarly, due to network space-efficiency, the PASIS configuration using $m = 3$ outperforms the minimal PASIS configuration (806 requests per second). Both PASIS and BFT have roughly the same network utilization per read operation (16 KB per operation). To be network-efficient, PASIS uses read witnesses and BFT uses "fast path" read operations. However, PASIS makes use of more storage-nodes than BFT does servers. As such, the aggregate bandwidth available for reads is greater for PASIS than for BFT, and consequently PASIS has a greater read throughput than

BFT. Although BFT could add servers to increase its read throughput, doing so would not increase its write throughput (indeed, write throughput would likely drop due to the extra inter-server communication).

BFT vs PASIS scalability summary

For PASIS and BFT, scalability is limited by either the server network utilization or server CPU utilization. Figure 3.10 shows that PASIS scales better than BFT in both. Consider write operations. Each BFT server receives an entire replica of the data, whereas each PASIS storage-node receives a data-fragment $\frac{1}{m}$ the size of a replica. The work performed by BFT servers for each write request grows with b. In PASIS, the server protocol cost decreases from 90 μs for $b = 1$ to 57 μs for $b = 4$, whereas in BFT it increases from 0.80 ms to 2.1 ms. The server cost in PASIS decreases because m increases as b increases, reducing the size of the data-fragment that is validated. We believe that the server cost for BFT increases because the number of messages that must be sent to all other servers increases.

3.6.4 Other results

Garbage collection

We assume a large window of storage version capacity, so garbage collection usually occurs during idle periods. But, even when it competes with real requests, garbage collection is inexpensive. Garbage collection requests are just batched read requests, except that no data need be returned for members that do not tolerate Byzantine clients. When Byzantine clients are tolerated, garbage collection must validate the cross checksum, which does require data-fragments.

Concurrency

Read-write concurrency can lead to client read operations observing repairable writes or aborting. To explore the effect of concurrency on the system, we measure multiclient throughput when accessing overlapping block sets. The experiment consists of four clients, each with four operations outstanding. Each client accesses a range of eight data blocks, some overlapping with other clients and some not, and no outstanding requests from the same client going to the same block.

At the highest concurrency level—all eight blocks in contention by all clients—we observed neither significant drops in throughput nor significant increases in mean response time. For example, the asynchronous repairable protocol member classified the initial candidate as complete 88.8% of the time, and found repair was necessary only 3.3% of the time. Since repair occurs rarely, the effect on average response time and throughput is minimal.

Impact of faults

Storage-node failures. For clients, storage-node failures have minimal impact on performance.

Client crash failures. Client crash failures appear as partially written data. Subsequent reads may observe these writes as incomplete or unclassifiable. If they are unclassifiable, the read must either abort or attempt repair. Repair adds much of the cost of performing a write, though, the round-trip to obtain a logical timestamp in an asynchronous system is not needed.

3.7 Discussion

3.7.1 Byzantine clients

In a storage system, Byzantine clients can write arbitrary values. The use of fine-grained versioning (e.g., self-securing storage [Strunk et al. 2000]) facilitates detection, recovery, and diagnosis from storage intrusions [Strunk et al. 2002]. Once discovered, arbitrarily modified data can be rolled back to its pre-corruption state.

Byzantine clients can also attempt to exhaust the resources available to the PASIS protocol. Issuing an inordinate number of write operations could exhaust storage space.

However, continuous garbage collection frees storage space prior to the latest complete write. If a Byzantine client were to intentionally issue incomplete write operations, then garbage collection may not be able to free up space. In addition, incomplete writes require read operations to roll-back behind them, thus consuming client computation and network resources. In practice, storage-based intrusion detection [Pennington et al. 2003] is probably sufficient to detect such client actions.

3.7.2 Timestamps from Byzantine storage-nodes

Byzantine storage-nodes can fabricate high timestamps that must be classified as incomplete by read operations. Worse, in each subsequent round of a read operation, Byzantine storage-nodes can fabricate more high timestamps that are just a bit smaller than the previous. In this manner, Byzantine storage-nodes can “attack” the performance of the read operation, but not its safety. To protect against such denial-of-service attacks, the read operation can consider all unique timestamps, up to a maximum of $b+1$, present in a *ResponseSet* as candidates before soliciting another *ResponseSet*. In this manner, each “round” of the read operation is guaranteed to consider at least one candidate from a correct storage-node and no more than b candidates from Byzantine storage-nodes.

3.7.3 Garbage collection

The proof of liveness (i.e., of wait-freedom) given in Appendix I assumes unbounded storage capacity. In practice, storage capacity is bounded; if storage capacity is exhausted, wait-freedom cannot be guaranteed. Prior experience indicates that it takes weeks of normal activity to exhaust the capacity of modern disk systems that version all write requests [Strunk et al. 2000].

Garbage collection is used to avoid storage exhaustion. In doing so, it can interact with concurrent read operations and concurrent write operations in such a manner that a read operation must be retried. Specifically a read operation could classify a concurrent write operation as incomplete, the write operation could then complete, and garbage collection could then delete all previous complete writes. If this occurs, the read operation's next round will observe an incomplete write with no previous history. Effectively, the read operation has "missed" the complete write operation that it would have classified as such. When it discovers this fact, the read operation retries (i.e., restarts by requesting a new *ResponseSet*). Thus, in theory, a read operation faced with perpetual write concurrency and garbage collection may never complete. In practice, such perpetual interaction of garbage collection and read-write concurrency for a given data-item is not realistic.

3.8 Summary

This chapter has developed an efficient Byzantine-tolerant protocol for reading and writing blocks of data by leveraging the versioning capabilities of storage-nodes. This protocol provides read–write semantics of full data blocks. As such, it is suitable as the basis for the data storage component within a survivable storage system. The subsequent chapters develop protocols that can provide more powerful read–modify–write semantics which are more suitable for constructing metadata services.

The R/W protocol is made space-efficient through the use of erasure codes and made scalable (in terms of faults tolerated) by offloading work from the storage-nodes to the clients. The protocol is work-efficient, since additional overheads only occur in cases of failures or read-write concurrency. Experiments demonstrate that PASIS, a prototype block storage system that uses the R/W protocol, scales well in the number of faults tolerated, supports 60% greater write throughput than BFT, and requires significantly less server computation than BFT.

4 Read/Conditional Write Block Protocol

Unlike data blocks that support read and write (R/W) operations only, metadata objects (e.g., directories), in order to preserve their integrity, require update operations that modify their existing contents, rather than those that blindly overwrite their previous contents. For example, two concurrent insertions into a directory using write operations can result in one being overwritten by the other. To support such operations, this chapter develops a *conditional write* (CW) operation that performs a write to an object only if the value of the object has not changed since the client last read it. As such, we refer to these objects as read/conditional write (R/CW) objects (vs. R/W). Moreover, read and conditional write operations are linearizable, thus ensuring atomicity for those that succeed.

The focus in this chapter is on techniques we have employed in the design of R/CW objects. The protocol is developed in the context of reading and writing full objects, or blocks (as was the R/W protocol). Since storage system data rarely requires the consistency provided by R/CW objects, the next chapter develops a metadata service based on an extension to this protocol. The extension provides a more general operation based interface that allows for finer-granularity access to objects.

Our R/CW protocols are designed around a hybrid fault model, in which different tolerances for Byzantine and benign (crash-recovery) failures can be specified, so that the cost of the protocol can be tuned to the number of each type of failure anticipated. The R/CW protocol, like the R/W protocol, is extremely optimistic: it is optimized for the common file system workload in which concurrent sharing is low and failures are rare. This optimism leads to a design for the R/CW protocols that, in the common case, requires just a single round of communication to perform a read operation and one additional round of communication for a conditional write (that can usually be optimized away). If client failures are encountered, or concurrency is observed, more rounds of communication may be necessary. As well, expensive cryptography, notably digital signatures, is avoided.

4.1 Overview

We describe the system in terms of N storage-nodes and an arbitrary number of clients and objects. Clients perform operations on objects. Storage-nodes host object replicas. Note, this is different from R/W objects which can use erasure coding for space-efficiency. The impact of using erasure coding is discussed in Section 4.6.1.

The R/CW protocol is comprised of *read operations* and *CW operations*. Both read operations and CW operations issue requests to sets of storage-nodes. CW requests are *executed* by storage-nodes. As in the R/W protocol, logical timestamps are used to totally order all CW operations and to identify CW requests from the same CW operation across storage-nodes. Each storage-node maintains a replica history, denoted *ReplicaHistory*, for each object it hosts. The replica history contains the entire set of CW requests executed on the object replica, ordered by logical timestamp (pruning the replica history is discussed in Section 4.3).

4.1.1 R/CW semantics

Before describing the R/CW protocol itself, this section describes the semantics achieved by the protocol. Conditional-write operations are a form of read–modify–write (RMW) operation. Read–modify–write semantics [Kruskal et al. 1988] are stronger than read–write semantics. More generally, as described in [Kruskal et al. 1988], a RMW operation is equivalent to the atomic execution of the follow function:

```
   RMW($X$, $f$)
1: $temp \leftarrow X$
2: $X \leftarrow f(X)$
3: return($temp$)
```

In this operation, the register X is read, an operation f is performed on X, and the resulting transformation is stored back in X. It has been shown that many other more powerful operations can be implemented as an RMW operation; e.g., test-and-set, fetch-and-add, etc. In a CW operation, an update of X is performed only if the value of X has not changed since it was previously read; thus, the write is conditioned-on the previously read value not having changed.

As in the R/W protocol, the protocol is optimistic, thus an operation may be complete, incomplete, or repairable. Every CW operation is preceded by a read operation that identifies the latest complete candidate. A CW operation is conditioned-on the latest complete candidate. R/CW semantics ensure that a single *conditioned-on chain* exists from any candidate back through all previous complete candidates to the initialized object. Therefore, no two complete writes can be conditioned on candidates such that the links formed between the complete write and the conditioned-on candidate overlap. Figure 4.1 more clearly illustrates the conditioned-on chain. R/CW semantics ensure that the conditioned-on time of the current candidate “points” at a complete candidate or at a well-known initial value. As well, since complete CW operations may only be observed as repairable, all repairable candidates must be repaired to maintain the integrity of the conditioned-on chain.

One of the main challenges of the R/CW protocol is to protect the integrity of the conditioned-on chain, especially from Byzantine clients. Byzantine clients can not be trusted to follow the protocol, thus they may attempt to break the conditioned-on chain by conditioning on an incorrect value (e.g., an incomplete CW operation or not the latest complete CW operation), see Figure 4.1(b). Briefly, to prevent these Byzantine attacks,

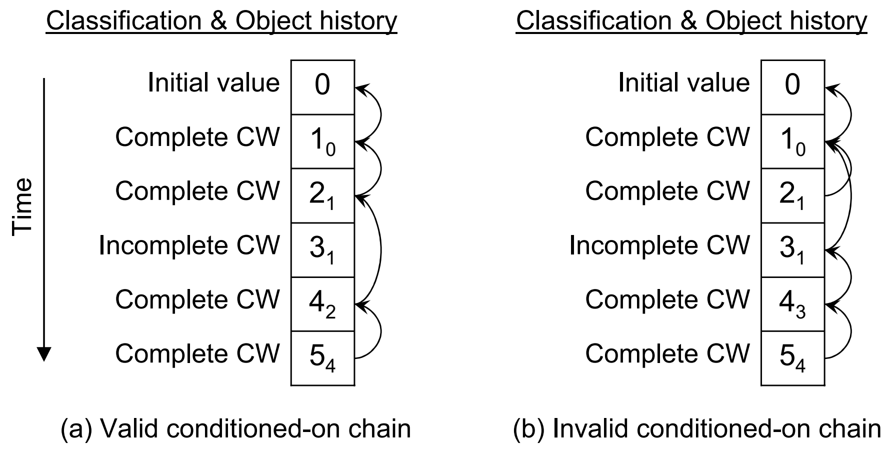


Figure 4.1: **Examples illustrating the *conditioned-on chain*.** *For each example, the complete object history and each version's classification for a CW object is shown. As well, the logical timestamp for each version is given, as is the logical time for the version on which it conditions (shown as a subscript to the logical timestamp). Note, the CW operation at logical time 3 is incomplete. Since both the version at logical time 2 and 3 condition on the version at time 1 (e.g., the may have been concurrent), only one version can complete successfully. Example (a) shows a valid conditioned-on chain. The version at logical time 3 is ignored. Example (b) shows an invalid conditioned-on chain. At logical time 4, a (Byzantine) client incorrectly conditions on the version at time 3 even though it is incomplete, thus corrupting the conditioned-on chain.*

clients must send "proof" with each CW operation supporting their action. Each client sends an *object history set* for each CW object being updated. The object history set contains the replica history of each storage-node that replied during the read phase. As well, each of the replica histories is "signed" (digital signatures may be used, but for performance we use *authenticators*, see Section 4.2.2) and acts as proof that the client is acting correctly. The "signed" object history set can then be validated by individually by each storage-node (this validation is discussed in detail in Section 4.3.3).

4.1.2 Read operation overview

At a high level, the R/CW protocol proceeds as follows. To perform a read operation, a client issues read requests to the set of storage-nodes. At least $N-t$ storage-nodes eventually respond to read requests. Due to the failure model and the asynchronous system model, only $N-t$ read responses can be collected by a client. The storage-node returns

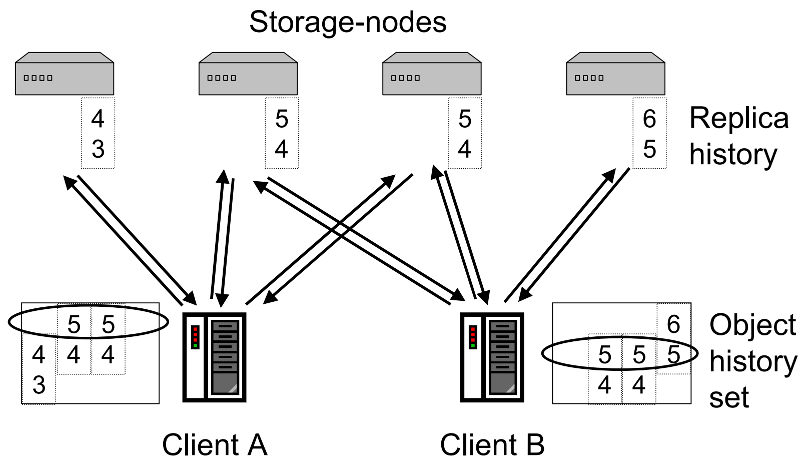


Figure 4.2: **Read requests and candidate classification.** *Clients A and B perform read operations in a system that tolerates one benign failure. Storage-nodes return object histories. Due to asynchrony, each client only receives responses from a subset of storage-nodes. Each client constructs an object history set from the object histories. In this example, client A classifies the candidate with logical timestamp 5 as repairable; client B classifies the candidate with logical timestamp 6 as incomplete, and 5 as complete.*

the replica history in response to a read request.

The client combines the replica histories returned by the storage-nodes into the object history set (denoted *ObjectHistorySet*). For example, *ObjectHistorySet*$[S]$ contains the replica history returned from storage-node S. Classification is performed on the timestamps within the object history set. The purpose of classification is to determine the timestamp of the latest complete (successful) update. A CW operation is complete once a threshold number of benign storage-nodes have executed CW requests. This threshold permits the R/CW protocol to ensure that no subsequent operation can return (or condition-on) a previous object value; as well, it defines classification. Classification identifies a *candidate*—a candidate is either *complete* or *repairable*. If the candidate is complete, then the read operation returns the object value associated with the candidate. If the candidate is repairable, the client performs a CW operation to repair the candidate. Once the repairable candidate is complete, its value is returned.

Aspects of the read operation are illustrated in Figure 4.2. In response to read requests, storage-nodes return object histories to the clients. In the example, each client constructs a different object history set since each client received responses from a different subset of storage-nodes. Classification is performed on the object history set. In the example, client *A* and *B* classify the candidate with logical timestamp 5 as repairable and complete respectively. Client *B* classified the candidate with logical timestamp 6 as incomplete prior to classifying 5 as complete. The exact rules for classification are given in Section 4.4.

4.1.3 CW operation overview

All CW operations are preceded by a read operation that identifies the candidate. Recall, a CW operation is conditioned-on the latest complete candidate (actually, on the object history set for which classification yields the candidate). R/CW semantics ensure that a single conditioned-on chain exists from any candidate back through all previous complete candidates to the initialized object. Each entry in a replica history is a ⟨logical timestamp, conditioned-on logical timestamp, value⟩ tuple. The elements of this tuple are denoted $\langle LT, LT_{\text{conditioned}}, Data\rangle$ and replica histories are initialized to $\langle \mathbf{0}, \mathbf{0}, \perp\rangle$.

The largest timestamp in the object history set is used by the client to create a timestamp for the CW operation. As discussed later, hashes of the object history set and the object value are also placed in the timestamp (these hashes ensure that all CW operations have a unique timestamp and protect against Byzantine entities). The client sends CW requests to all storage-nodes. The CW request contains the timestamp of the CW operation, the object history set constructed by the preceding read operation, the candidate (found from classification of the object history set), and the object value.

Correct storage-nodes execute a CW request only if the timestamp, value, and replica history can all be validated. Validation ensures failure atomicity, concurrency atomicity, and, as discussed below, protects against Byzantine entities. If sufficient storage-nodes execute CW requests, the CW operation completes; otherwise, it aborts. CW operations may abort due to concurrency. Since repair is a CW operation, and may be part of a read operation, read operations may also abort.

To ensure that only one CW operation completes that is conditioned upon another CW operation, a *barrier* may be written to ensure that outstanding concurrent CW operations cannot complete. Barriers mark a point in time without inserting a value into the system. Thus, a completed barrier written infront of an incomplete value prevents (or bars) the incomplete operation from ever completing; storage-node validation will fail since the barrier's timestamp is larger than the timestamp being conditioned on by the incomplete operation. Barriers allow CW operations to be issued that may complete in the face of concurrency or client failure.

4.2 Mechanisms

This section describes various mechanisms employed to guarantee safety within the the R/CW protocol.

4.2.1 Validating timestamps

Logical timestamps are structured values with three members: the primary timestamp (*Time*), the object history set verifier (*Verifier_OHS*), and the value verifier (*Verifier_Data*). The verifiers are collision-resistant hashes over the object history set and the object's value respectively. In comparing two logical timestamps, to determine which is greater, the major timestamps are first compared, and then the verifiers are compared. Since verifiers are guaranteed to be unique (for unique object values) all timestamps are guaranteed to be unique.

The use of the collision-resistant hash cryptographic primitive protects against Byzantine entities. Byzantine storage-nodes cannot undetectably corrupt values written to them, because the hash of the object's value is in the timestamp. Byzantine clients cannot perform *poisonous writes* [Martin et al. 2002]. In a poisonous write, a Byzantine client writes different values to different storage-nodes with the same timestamp. Storage-nodes validate the value sent in a CW request with the value verifier. Since in the R/CW protocol client's transmit full replicas, as opposed to erasure coded fragments in the R/W protocol, storage-node validation prevents poisonous writes. As well, validation ensures that a Byzantine client cannot make a correct storage-node appear Byzantine (i.e., only Byzantine storage-nodes return values that cannot be validated by the client).

4.2.2 Authenticators

To ensure R/CW semantics, the conditioned-on relationship between the value in the CW request and the candidate in the object history set must be maintained. This conditioned-on relationship is validated by a storage-node before it executes a CW request. For a storage-node to validate the conditioned-on relationship, it must “know” that the replica histories in the object history set are indeed those returned by other storage-nodes.

Generally, digital signatures are used for such purposes. However, digital signatures are computationally expensive to compute and verify. Keyed cryptographic hash functions can be evaluated approximately three orders of magnitude faster than digital signatures. To make Byzantine fault-tolerant agreement efficient, Castro and Liskov used *authenticators* in lieu of digital signatures in BFT [Castro and Liskov 1998b]. Authenticators are vectors of keyed hashes: the i^{th} element in the vector is used to prove the authenticity of the message to entity i. To enable authenticators, all pairs of entities that need to prove message authenticity to one another must share distinct secret keys. Authenticators are not as strong a primitive as signatures: any entity can verify an entire signature, whereas only entities in the vector of keyed hashes can validate the authenticator (and, at that, only its own entry in the authenticator).

Authenticators are used by storage-nodes in the R/CW protocol to “sign” replica histories returned in read responses. If authenticator validation fails, the storage-node cannot tell if a Byzantine client corrupted a valid replica history, or if a Byzantine storage-node constructed an invalid replica history, since the object history set is constructed by the client. Invalid authenticators are discussed in Section 4.6.2.

During repair, authenticators allow clients with read-only access to an object to perform repair on the object. In many systems, there is an asymmetry between write privileges and read privileges. Authenticators provide sufficient proof to storage-nodes that the object history set is valid and thus that some candidate is repairable. Consequently, storage-nodes can execute “repair” CW requests from read-only clients.

4.3 Protocol

This section pseudo-code for classifying candidates, performing conditional write operations, and validating CW requests. As well, read operations and storage-node actions are discussed in detail.

4.3.1 Read operation

Figure 4.3 shows the pseudo-code for the read operation. The read operation begins by issuing read requests to the set of N storage-nodes. Given the asynchronous nature of the protocol, and the crash-recovery failure model for storage-nodes, no more than $N-t$ read responses are collected.

In response to a read request, the storage-node returns its replica history. A client can explicitly request a specific version of the object (based on candidate classification), and the storage-node returns its value. This functionality is used to implement *read witnesses*: only one storage-node need return the value of its object replica, the replica histories of other storage-nodes act as witnesses [Pâris 1986] that validate the correctness of the value (through the object's value hash).

The client uses the returned replica histories to construct the *ObjectHistorySet* (cf. line 100). Recall, each entry in a replica history is a $\langle$logical timestamp, conditioned-on logical timestamp, value$\rangle$ tuple. For simplicity of presentation, data corresponding to each entry in the replica history is returned. However, in practice, only the data value associated with the latest logical timestamp is returned. Optimistically, the latest timestamp is usually classified as complete and this data is sufficient, otherwise an extra read round-trip is required to fetch the appropriate data version (as in the R/W protocol).

A storage-node also attaches an authenticator to the replica history it returns. When the client constructs the object history set, *ObjectHistorySet*, each replica history in the set has an authenticator. The client can cache the object history set and authenticators and use them in a future CW request.

The pseudo-code for the CW `DO_READ` operation is very similar to the code shown

```
READ() :
100: $ObjectHistorySet$ := DO_READ()
101: $\langle Candidate, Status\rangle$ := CLASSIFY($ObjectHistorySet$)
102: if ($Status$ = CLASSIFIED_COMPLETE) then
103:     return (SUCCESS, $\langle Candidate.LT, Candidate.Data\rangle$)
104: else
105:     /* $Status$ = CLASSIFIED_REPAIRABLE */
106:     return (CONDITIONAL_WRITE($Candidate$, $Candidate.LT_{\text{conditioned}}$, $Candidate.Data$, $ObjectHistorySet$))
107: end if

DO_READ() :
200: $ResponseSet := \emptyset$
201: repeat
202:     for all $S \in \{S_1, \dots, S_N\} \setminus ResponseSet.S$ do
203:         SEND($S$, READ_REQUEST)
204:     end for
205:     if (POLL_FOR_RESPONSE() = TRUE) then
206:         $\langle S, ReplicaHistory\rangle$ := RECEIVE_READ_RESPONSE()
207:         if (($S \notin ResponseSet.S$) AND VALIDATE($ReplicaHistory$) = SUCCESS)) then
208:             $ObjectHistorySet[S] := ReplicaHistory$
209:             $ResponseSet := ResponseSet \cup \langle S\rangle$
210:         end if
211:     end if
212: until ($|ResponseSet| = N - t$)
213: return ($ObjectHistorySet$)

CLASSIFY($ObjectHistorySet$) :
300: $\langle Candidate, Count\rangle$ := SELECT_CS($ObjectHistorySet$, $\perp$)
301: loop
302:     if ($Count \geq$ COMPLETE) then
303:         return ($\langle Candidate$, CLASSIFIED_COMPLETE$\rangle$)
304:     else if ($Count \geq$ INCOMPLETE) then
305:         return ($\langle Candidate$, CLASSIFIED_REPAIRABLE$\rangle$)
306:     end if
307:     /* Incomplete candidate: find new candidate and loop again. */
308:     $\langle Candidate, Count\rangle$ := SELECT_CS($ObjectHistorySet$, $Candidate.LT$)
309: end loop
```

Figure 4.3: **Client read and classification pseudo-code.**

in the R/W protocol chapter (Figure 3.6). First, the DO_READ function discards any responses that cannot be validated (cf. line 207). Since data is assumed to be replicated, the value verifier (stored in the timestamp) is simply the hash over the data. Second, the *ObjectHistorySet* is constructed from the set of storage-node responses (cf. line 208).

The client then performs classification on the object history set (cf. line 101). The pseudo-code for classification is also shown in Figure 4.3. The function iteratively identifies and classifies potential candidates until a valid complete or repairable candidate is found. The function SELECT_CS, on line 300, identifies the potential candidate with the highest timestamp, and sets *Candidate* accordingly. The value *Count* returned from

SELECT_CS is the count of the number of storage-nodes in the *ObjectHistorySet* that host the potential candidate. Note, only candidates that can be validated with the value verifier in the timestamp are chosen; as well, barrier-writes are ignored.

Once a potential candidate has been chosen, it is classified as either complete, repairable, or incomplete. To classify a potential candidate, *Count* is compared with the constants COMPLETE and INCOMPLETE. The derivation of these constants are described in Section 4.4.

If the potential candidate is classified as complete, the read operation returns the value associated with the candidate. If the potential candidate is classified as incomplete, SELECT_CS is called again, but with the potential candidate's timestamp (cf. line 308). A new potential candidate, with a lower timestamp, is identified. Candidate classification begins again.

If the potential candidate is classified as repairable, the client performs repair. It does so by issuing a CW operation (shown in Figure 4.5) with the value of the repairable candidate, see line 106 in Figure 4.3. If the repairable candidate contains the latest timestamp observed by the client (i.e., its $LT = \texttt{MAX}[\mathit{ObjectHistorySet}]$), then the client can attempt repair by completing the operation at repairable's logical time. Otherwise, a barrier is needed to block any competing incomplete operations from completing. When a barrier is needed, a new (higher) logical timestamp is generated. The condition on timestamp ($LT_{\text{conditioned}}$) is set to the timestamp of the latest complete write. The latest complete write is the CW conditioned on by the repairable candidate. If the repair operation fails, then the read operation aborts and must be retried. An example of a repair that requires a barrier-write is shown in Figure 4.4.

4.3.2 CW operation

The pseudo-code for the CW operation is shown in Figure 4.5. First, the logical timestamp for the CW operation, LT, is constructed. The major part of LT is determined by incrementing the major part of the largest timestamp in the *ObjectHistorySet*. The verifiers are determined by taking the hash of the object history set and the object value for

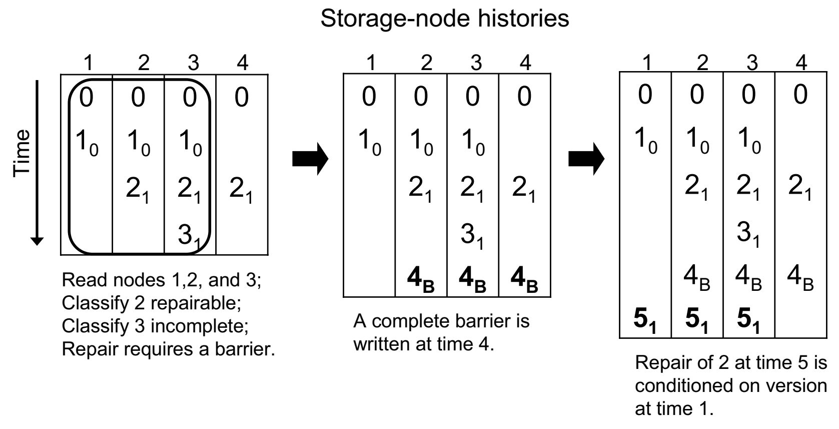

Storage-node histories
Time
1 2 3 4
Read nodes 1,2, and 3;
Classify 2 repairable;
Classify 3 incomplete;
Repair requires a barrier.
A complete barrier is written at time 4.
Repair of 2 at time 5 is conditioned on version at time 1.

Figure 4.4: **Example of repair requiring a barrier-write.** *For this setup:* $N = 4, \texttt{COMPLETE} = 3, \texttt{INCOMPLETE} = 2$*. The object history is shown at 4 storage-nodes during the progression of a read operation. The subscript to each version's timestamp corresponds to the timestamp on which the version is conditioned. First, the client classifies LT* 2 *as repairable (although it is actually complete). However, an incomplete operation exists at LT* 3*. To block this operation from completing, a barrier is written at LT* 4*. Finally, repair is attempted at* $LT = 5$*,* $LT_{\text{conditioned}} = 1$*. Note, that the condition on time is the same as the repairable candidate's.*

the CW operation. It is important to note that every logical timestamp generated is at least 1 larger than the latest complete write that currently exists in the system.

Before the CW operation is issued with the value, the object history set is checked to see if a *barrier* is needed (cf. line 407). Barriers ensure that multiple CW operations conditioned-on the same candidate do not complete. A barrier is needed if any of the replica histories in the object history set contain timestamps that are greater than that of the candidate's timestamp (and are not themselves barriers). If such a timestamp exists, then there may be another CW operation concurrent to this CW operation. If the barrier completes, then the concurrent CW operation cannot complete.

To create a barrier, the client performs a DO_WRITE with the $\perp$ value, a $\perp$ verifier, and *LT.Barrier* set to TRUE (called a *barrier-write*). This allows the CW operation to be executed at storage-nodes that host its barrier. The DO_WRITE function issues CW requests to all storage-nodes. The object history set is updated with each response DO_WRITE receives, line 508. As in DO_READ, the replica history is first validated. However, one should note that even if the operation failed, the replica history is returned. This allows the client

```
CONDITIONAL_WRITE($Candidate, LT_{conditioned}, Data, ObjectHistorySet$) :
400: /* Construct the logical timestamp for the CW operation. */
401: $LT_{latest}$ := MAX_TIMESTAMP($ObjectHistorySet$)
402: $LT.Time := LT_{latest}.Time + 1$
403: $LT.Verifier\_OHS$ := HASH($ObjectHistorySet$)
404: $LT.Verifier\_Data$ := HASH($Data$)
405: $LT.Barrier$ := FALSE
406: /* If necessary, write a barrier. */
407: if ($LT_{latest} > Candidate.LT$) then
408:     $LT_{barrier} := LT$
409:     $LT_{barrier}.Verifier\_Data := \perp$
410:     $LT_{barrier}.Barrier$ := TRUE
411:     $\langle Count, ObjectHistorySet \rangle$ := DO_WRITE($LT_{barrier}, LT_{conditioned}, ObjectHistorySet, \perp$)
412:     if ($Count <$ COMPLETE) then
413:         return (ABORT, $\langle \perp, 0 \rangle$)
414:     else
415:         /* Re-classify based on returned object history set*/
416:         $\langle Candidate\_new, Status \rangle$ := CLASSIFY($ObjectHistorySet$)
417:         if ($Candidate\_new \neq Candidate$) then
418:             /* Abort if classification yields different result */
419:             return (ABORT, $\langle \perp, 0 \rangle$)
420:         end if
421:     end if
422: end if
423: /* Perform the CW operation. */
424: $\langle Count, ObjectHistorySet \rangle$ := DO_WRITE($LT, LT_{conditioned}, ObjectHistorySet, Data$)
425: if ($Count <$ COMPLETE) then
426:     return (ABORT, $\langle \perp, 0 \rangle$)
427: end if
428: return (SUCCESS, $\langle Data, LT \rangle$)


DO_WRITE($LT, LT_{conditioned}, ObjectHistorySet, Data$) :
500: $ResponseSet := \emptyset$
501: repeat
502:     for all $S \in \{S_1, \ldots, S_N\} \setminus ResponseSet.S$ do
503:         SEND($S$, WRITE_REQUEST, $LT, LT_{conditioned}, ObjectHistorySet, Data$)
504:     end for
505:     if (POLL_FOR_RESPONSE() = TRUE) then
506:         $\langle S, ReplicaHistory, Status \rangle$ := RECEIVE_WRITE_RESPONSE()
507:         if (($S \notin ResponseSet.S$) AND VALIDATE($ReplicaHistory$) = SUCCESS)) then
508:             $ObjectHistorySet[S] := ReplicaHistory$
509:             $ResponseSet := ResponseSet \cup \langle S \rangle$
510:             if ($Status$ = SUCCESS) then
511:                 $Count := Count + 1$
512:             end if
513:         end if
514:     end if
515: until ($|ResponseSet| = N - t$)
516: return ($\langle Count, ObjectHistorySet \rangle$)
```

Figure 4.5: **Client-side CW operation pseudo-code.**

to retry a write without performing a read to obtain the latest object history. Once $N - t$ responses have been received (line 515), DO_WRITE returns.

Once the barrier-write has completed successfully it is necessary to perform classi-

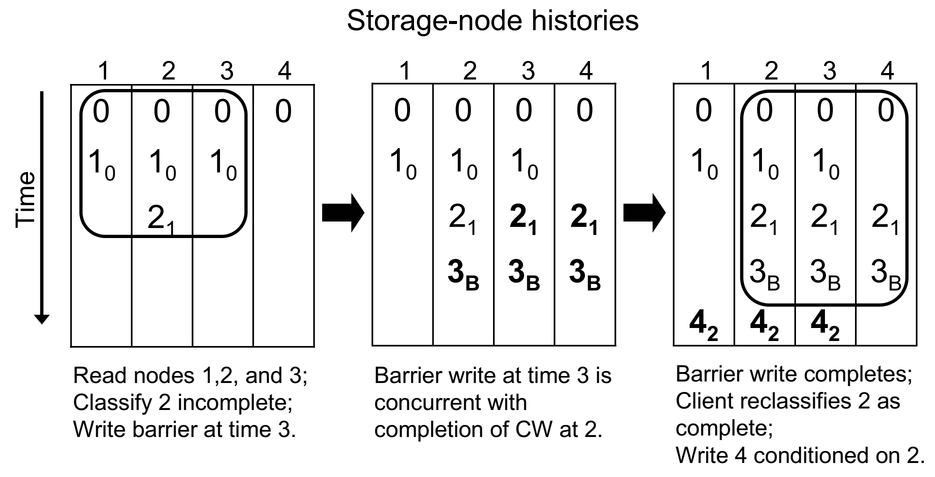

Storage-node histories
Time
1 2 3 4
0 0 0 0
1_0 1_0 1_0
2_1
3_B
4_2
Read nodes 1,2, and 3; Classify 2 incomplete; Write barrier at time 3.
Barrier write at time 3 is concurrent with completion of CW at 2.
Barrier write completes; Client reclassifies 2 as complete; Write 4 conditioned on 2.

Figure 4.6: **Example of a barrier-write.** *For this setup:* $N = 4, \texttt{COMPLETE} = 3, \texttt{INCOMPLETE} = 2$. *The object history is shown at 4 storage-nodes during the progression of a CW operation. First, the client classifies LT* 2 *as incomplete. Next, it attempts to write a barrier at* $LT = 3$*, however the barrier-write is concurrent with the completion of* 2*. Once the barrier-write completes, the client re-classifies LT* 2 *as complete. The CW operation is aborted and retried at* $LT = 4, LT_{\text{conditioned}} = 2$.

fication over the new object history set, line 416. This is necessary, since barrier-writes may be executed at storage-nodes hosting the CW operation that they are trying to block (i.e., an incomplete operation may complete just prior to the execution of the barrier). If classification yields the same candidate (cf. line 417), the client can perform the value-write. And, if the value-write completes, then the *Candidate* is returned (in case the CW operation is called for repair). If either the barrier-write or value-write do not complete, then the CW operation aborts, and must be retried (including the read phase). Figure 4.6 shows an example of a barrier-write that necessitates re-classification.

4.3.3 CW requests at storage-nodes

Receiving CW requests

Pseudo-code describing the reception of a CW request at a storage-node is shown in Figure 4.7. First, the request must be validated. Due to the complexity of validation, it is described fully in the next sub-section. If validation succeeds, the request is inserted

```
RECEIVE_WRITE_REQUEST($LT$, $LT_{\text{conditioned}}$, $ObjectHistorySet$, $Data$) :
600: if (VALIDATE_CW_REQ($LT$, $LT_{\text{conditioned}}$, $ObjectHistorySet$, $Data$)) then
601:    /* Execute the CW request */
602:    $Request := \langle LT, LT_{\text{conditioned}}, Data \rangle$
603:    $ReplicaHistory := ReplicaHistory \cup Request$
604:    /* Prune history */
605:    if ($Data \neq \perp$) then
606:       PRUNE_HISTORY_SET($ReplicaHistory$, $LT_{\text{conditioned}}$)
607:    end if
608:    SEND(WRITE_RESPONSE, $S$, $ReplicaHistory$, SUCCESS)
609:    return
610: end if
611: SEND(WRITE_RESPONSE, $S$, $ReplicaHistory$, FAIL)
612: return
```

Figure 4.7: **Reception of a CW at storage-node *S*.**

into the storage-node's local history, line 603. The request is comprised of the tuple: $\langle LT, LT_{\text{conditioned}}, Data \rangle$.

If an object history set contains a complete candidate, the storage-node can prune its replica history up to the complete candidate's timestamp (cf. line 606). A validated object history set always contains a candidate that is complete (although such a candidate may be earlier in the object history set than a repairable candidate returned from classification); note, the initial timestamp 0 can be considered the earliest complete candidate. All versions prior to the latest complete (non-barrier) CW (i.e., prior to the CW's $LT_{\text{conditioned}}$) can be pruned; however, no pruning occurs on barrier-writes. The use of authenticators obviates the need for distributed garbage collection as was necessary in the RW protocol. Finally, a response is sent back to the client. Note that the storage-node's local history is always transmitted in the response. This allows the client to update the object history set and re-perform classification if necessary (e.g., if the CW operation failed).

Validating CW requests

Storage-nodes must validate CW requests. If validation succeeds, the storage-node executes the CW request: it executes the value-write and updates its replica history to include the CW request. Remember, the replica history is stored in stable storage that endures the crash-recovery cycle. If validation fails, the storage-node rejects the CW request.

With the object history set, the storage-node can perform the exact same logic as the

```
VALIDATE_CW_REQ($LT, LT_{\text{conditioned}}, ObjectHistorySet, Data$) :
/* Validate LT.Time, ensures logical time is always increasing */
if ($LT.Time \neq$ ((MAX_TIMESTAMP[$ObjectHistorySet$]).$Time + 1$)) then
    return (FAIL)
end if
/* Validate authenticators for each history set */
if (VALIDATE_AUTHENTICATORS($ObjectHistorySet$) = FAIL) then
    return (FAIL)
end if
/* Validate verifiers */
if (HASH($Data$) $\neq LT.Verifier\_Data$) then
    return (FAIL)
else if (HASH($ObjectHistorySet$) $\neq LT.Verifier\_OHS$) then
    return (FAIL)
end if

/* Perform classification to find latest logical timestamps */
$\langle LT_{\text{latest\_comp\_nb}}, LT_{\text{latest\_comp\_nb\_bd}}, LT_{\text{latest\_nb}},$
    $LT_{\text{latest\_barrier}}, LT_{\text{latest\_ni\_nb}}, Classify_{\text{latest\_barrier}} \rangle :=$ CLASSIFY($ObjectHistorySet$)

/* Check if a barrier is needed, and not writing a barrier */
if (($LT_{\text{latest\_comp\_nb\_bd}} \neq LT_{\text{latest\_nb}}$) AND
      (($LT_{\text{latest\_barrier}} < LT_{\text{latest\_nb}}$) OR ($Classify_{\text{latest\_barrier}} \neq$ COMPLETE)) then
    /* Barrier is needed, make sure this is a barrier */
    if ($LT.Barrier$ = FALSE) then
        return (FAIL)
    end if
else if ($LT.Barrier$ = TRUE) then
    return (FAIL)
end if

/* Validate replica acceptance policy */
if (MAX[$ReplicaHistory$] > MAX[$LT_{\text{latest\_nb}}, LT_{\text{latest\_barrier}}$]) then
    return (FAIL)
end if

if ($LT.Barrier$ = FALSE) then
    /* Validate condition on relationship, conditioning on a complete*/
    if (($LT_{\text{latest\_nb}} = LT_{\text{latest\_comp\_nb}}$) AND ($LT_{\text{conditioned}} \neq LT_{\text{latest\_comp\_nb}}$)) then
        /* Classified complete as latest timestamp, but $LT_{\text{conditioned}}$ not conditioned on latest complete */
        return (FAIL)
    end if

    /* Validate condition on relationship, performing repair */
    if ((($LT_{\text{latest\_nb}} \neq LT_{\text{latest\_comp\_nb}}$) AND ($LT_{\text{latest\_nb}} = LT_{\text{latest\_ni\_nb}}$)) AND
          ($LT.Verifier\_Data \neq LT_{\text{latest\_ni\_nb}}.Verifier\_Data$) OR ($LT_{\text{conditioned}} \neq LT_{\text{latest\_comp\_nb\_bd}}$)) then
        return (FAIL)
    end if
end if
return (SUCCESS)
```

Figure 4.8: **Validation of a CW request at storage-node *S*.**

client, and validate that the client is acting correctly. The pseudo-code for validation is shown in Figure 4.8.

First, a sanity check is performed on the primary time within the logical timestamp, Logical timestamps returned from classifying the object history set on the storage-node:

- $LT_{\text{latest\_barrier}}$: Latest barrier logical timestamp;
- $LT_{\text{latest\_nb}}$: Latest non-barrier logical timestamp;
- $LT_{\text{latest\_ni\_nb}}$: Latest non-incomplete, non-barrier logical timestamp;
- $LT_{\text{latest\_comp\_nb}}$: Latest complete, non-barrier logical timestamp;
- $LT_{\text{latest\_comp\_nb\_bd}}$: Latest complete, non-barrier, by deduction logical timestamp.
  Note: $LT_{\text{latest\_comp\_nb\_bd}} = \texttt{MAX}[LT_{\text{latest\_comp\_nb}}, LT_{\text{latest\_ni\_nb}_{\text{conditioned}}}]$

Figure 4.9: **Logical timestamps returned from classification.**

line 701. Next, the authenticators for the object history set is validated (cf. line 705). Recall, the object history set is comprised of history sets from each of the storage-nodes queried during the read operation phase. Each of these history sets has a corresponding authenticator that must be validated. The impact of failed authenticator validation is discussed in Section 4.6.2.

Next, the verifiers, stored within the timestamp, are validated (cf. lines 708 - 713). Again, the storage-node is replicating client logic to ensure that the validating timestamp is well-formed. The object history set verifier determines the conditioned-on relationship for a the CW operation and the value verifier determines the value of the CW operation, thus limiting the actions a Byzantine client can perform.

By performing classification on the object history set, line 716, the storage-node can validate that the candidate is the correct candidate, and that the timestamp is the correct timestamp (since both are deterministic given an object history set). As well, the storage-node can check to see if a *barrier* is required (cf. line 719). To make these checks, storage-node classification returns multiple logical timestamps, each set to a logical timestamp within the object history set. These timestamps are described within Figure 4.9.

The descriptions of most of the timestamps returned from classification are clear. However, the $LT_{\text{latest\_comp\_nb\_bd}}$ timestamp deserves further discussion. As described, this timestamp represents a version that is complete by deduction. A candidate that is complete by deduction is a candidate that is necessarily complete, but has not been observed as such. This occurs when a repairable (non-barrier) candidate *A* is observed (in the object history set) to have conditioned upon another, earlier, repairable (non-barrier) candidate *B*. Recall, a complete candidate may be viewed as repairable. Since *A* is repairable, it has passed validation at $\geq Q_C - t$ storage-nodes, of which at least $Q_C - t - b$ are benign (see the constraint derivations for more details, Section 4.4). The only possible way that *A* can pass validation at a benign storage-node is by conditioning on a complete (non-barrier) candidate. Thus, *B* must be complete (by deduction).

Barriers are required to squash pending CW requests (e.g., incomplete CW operations from failed clients). A barrier is required if there exists any non-barrier, non-complete CW request that exists within the object history set at a later logical timestamp than any other complete CW write or barrier operation (i.e., there exists a pending incomplete CW operation). If the *Barrier* portion of the logical timestamp is set to `TRUE`, the storage-node knows that the CW request is part of a barrier-write. The only further validation that occurs for barrier-writes is that of the *acceptance-check* against the replica's history.

The acceptance-check, line 728 ensures that a CW request has not been executed locally that is later than the latest logical timestamp present within the object history set.

If the CW is not a barrier-write, the conditioned-on relationship is then validated (cf. lines 734 - 743). There are two cases for validation. First, a complete candidate is classified as the latest non-barrier. In this case, the conditioned-on timestamp is verified against the latest complete, non-barrier CW (cf. line 735).

Second, a repairable candidate is classified as the latest non-barrier. That is, the latest non-barrier is the same as the latest non-incomplete, non-barrier operation and the latest non-barrier is not the latest complete, line 741. If the candidate is repairable, two validations are performed. First, the storage-node validates that the object value for the CW request is the same as the repairable candidates (this ensures that the repair occurs correctly). Second, the CW's conditioning on timestamp is checked against the repairable's conditioning on timestamp (note, in this case $LT_{\text{latest\_comp\_nb\_bd}} = LT_{\text{latest\_ni\_nb}_{\text{conditioned}}}$); this ensures the continuity of the conditioning chain.

A summary of the conditions that hold if storage-node validation succeeds is presented in Figure 4.10. These are derived from the pseudo-code shown in Figure 4.8.

Postulates that hold if validation succeeds (3 and 4 only hold if $LT.Barrier$ is FALSE):

i $(LT_{\text{latest\_comp\_nb\_bd}} = LT_{\text{latest\_nb}})$ OR $((LT_{\text{latest\_barrier}} > LT_{\text{latest\_nb}})$ AND $(Classify_{\text{latest\_barrier}} = \texttt{COMPLETE}))$

ii $\texttt{MAX}[ReplicaHistory] \leq MAX[LT_{\text{latest\_nb}}, LT_{\text{latest\_barrier}}]$

iii **if** $(LT_{\text{latest\_nb}} = LT_{\text{latest\_comp\_nb}})$ **then** $LT_{\text{conditioned}} = LT_{\text{latest\_comp\_nb}}$

iv **if** $((LT_{\text{latest\_nb}} \neq LT_{\text{latest\_comp\_nb}})$ AND $(LT_{\text{latest\_nb}} = LT_{\text{latest\_ni\_nb}}))$ **then**
$(LT.Verifier\_Data = LT_{\text{latest\_ni\_nb}}.Verifier\_Data)$ AND $(LT_{\text{conditioned}} = LT_{\text{latest\_comp\_nb\_bd}})$

Figure 4.10: **Validation postulates.**

4.4 Constraints

This section presents bounds on N and Q_C, the definition of a complete CW operation (in terms of t and b), and constraints on COMPLETE and INCOMPLETE. The derivation of constraints is similar to that for the protocol for R/W objects in Section 3.4. The differences in bounds arise mainly from the added constraint that repairable candidates must intersect with complete candidates for the R/CW protocol to be safe. A formal proof of the safety and liveness of the R/CW protocol and its extension to the Q/U protocol is given in [Abd-El-Malek et al. 2004].

4.4.1 Read classification rules

Recall that the *candidate* is the data-item version, returned by a read request, with the greatest logical timestamp. The set of read responses that share the candidate's timestamp are the *candidate set*. Constraints on COMPLETE and INCOMPLETE are required to ensure two properties. The first property is that if a candidate is ever classified as complete, then any subsequent read operation observes the complete candidate as repairable. The second property is that if candidate A is complete and conditioned-on candidate B, any repairable candidate with a timestamp greater than A either conditioned-on A or can traverse the conditioned-on chain back to A.

To classify a candidate as complete, a candidate set of at least Q_C benign storage-nodes must be observed. In the worst case, at most b members of the candidate set may be Byzantine, thus,

$$|CandidateSet| - b \geq Q_C \Rightarrow \texttt{COMPLETE} = Q_C + b. \tag{4.1}$$

To classify a candidate as incomplete, the candidate must be incomplete (i.e., fewer than Q_C benign storage-nodes have executed the CW). We consider a rule for classifying incomplete candidates that takes advantage of $N-t$ responses from storage-nodes. In the crash-recovery model, eventually, a client is guaranteed to receive this many responses—even though, there may be periods during which more than t storage-nodes are crashed. Moreover, a client cannot expect more than this many responses, since up to t storage-nodes may never recover (and in an asynchronous environment crash failures are undetectable). Thus, the rule for classifying a candidate incomplete is,

$$|CandidateSet| + t < Q_\mathrm{C} \Rightarrow \mathtt{INCOMPLETE} = Q_\mathrm{C} - t. \tag{4.2}$$

Candidates that cannot be classified as complete or incomplete are classified as repairable.

Given these constraints on COMPLETE and INCOMPLETE, consider the examples illustrated in Figure 4.2. In this example, $t = 1$ and $b = 0$, so $\mathtt{COMPLETE} = 3$, $\mathtt{INCOMPLETE} = 2$, and $N = 4$. Since client A observes a candidate with timestamp 5 in two replica histories, and this is not less than the incomplete threshold, it classifies 5 as repairable. Whereas candidate 6 is observed by client B in one replica history and classified incomplete.

4.4.2 Real repairable candidates

This property ensures that colluding Byzantine storage-nodes are unable to fabricate a candidate that a correct client deems repairable. To achieve this property, a candidate set of size b must be classifiable as incomplete. Substituting $|CandidateSet| = b$ into (4.2),

$$b + t < Q_\mathrm{C}. \tag{4.3}$$

4.4.3 Complete candidate—repairable candidate intersection

This property prevents multiple CW operations conditioned on the same candidate (i.e., with the same $LT_\mathrm{conditioned}$) performed by correct clients from completing. A complete candidate cannot be allowed to co-exist with a repairable candidate. Since a complete candidate may be observable as repairable, a client may observe two repairable candidate's (even though one is complete) and not know which candidate to repair. If the "wrong candidate" (i.e., the one that is not complete) is repaired, the condition on chain is violated.

To achieve this property, it is necessary that a complete candidate and repairable candidate intersect at at least one benign storage-node. Thus, from the lower bounds of repairable and complete candidates,

$$\begin{aligned} \texttt{COMPLETE} + \texttt{INCOMPLETE} &> N + b, \\ Q_C + b + Q_C - t &> N + b, \\ 2Q_C &> N + t. \end{aligned} \tag{4.4}$$

4.4.4 Read set intersection

The intersection between a complete and a candidate set of Q_C benign storage-nodes must result in at least a repairable being observed. Thus,

$$\begin{aligned} N + \texttt{INCOMPLETE} &\le Q_C + \texttt{COMPLETE}, \\ N + Qc - t &\le Q_C + (Q_C + b), \\ N &\le Q_C + b + t. \end{aligned} \tag{4.5}$$

4.4.5 CW termination

A CW operation is defined to be complete once a total of Q_C benign storage-nodes have executed the CW.

There must be sufficient good storage-nodes in the system for a CW operation by a correct client to complete. A client must terminate after it receives $N - t$ responses. As well, up to b responses may be from Byzantine storage-nodes (who lie about executing the operation), similar to the R/W protocol (see Section 3.4.4).

However, there is another action a Byzantine storage-node can take that was not possible in the R/W protocol. A Byzantine storage-node can lie that it could not execute the

CW request because validation failed (e.g., it hosts a CW request with a timestamp greater than *LT*), see Section 4.3.3. This action requires that a CW operation can complete in the face of *b* storage-nodes that reject the operation. If the reject message always reaches the client before some accept message from a benign storage-node, the client will always abort the CW operation (since in an asynchronous crash-recovery model it cannot await all responses). In the case that Byzantine storage-nodes do not control the network, then, probabilistically, the CW operation will eventually successfully complete.

Thus, for the CW operation to be guaranteed to complete (i.e., for a client to ensure that it is possible for Q_C benign storage-nodes to execute CW requests during the CW operation),

$$\begin{aligned} Q_C + b &\le N - t - b, \\ Q_C &\le N - t - 2b. \end{aligned} \tag{4.6}$$

4.4.6 Constraint summaries

Constraints (4.4) and (4.6) lead to a constraint on Qc that supersedes (4.3):

$$\begin{aligned} Q_C + t + 2b \le N &< 2Q_C - t, \\ Q_C + t + 2b &< 2Q_C - t, \\ 2t + 2b &< Q_C. \end{aligned} \tag{4.7}$$

Adding (4.4) and (4.5) leads to a constraint on N:

$$\begin{aligned} 2N &< 3Q_C + b, \\ N &< \frac{3Q_C + b}{2}. \end{aligned} \tag{4.8}$$

And so, the overall constraints on Q_C and N can be written as,

$$2t+2b+1 \le Q_\mathrm{C} \le N-t-2b;$$
$$Q_\mathrm{C}+t+2b \le N < \mathtt{MIN}\left[2Q_\mathrm{C}-t,\ \frac{3Q_\mathrm{C}+b}{2}\right].$$

Which leads to the minimal bounds of:

$$Q_\mathrm{C} = 2t+2b+1;$$
$$N = 3t+4b+1.$$

4.4.7 Improving the bounds on N and Q_C

If we relax the liveness guarantee provided in 4.4.5, the bounds on N and Qc can be significantly improved. Recall, the bounds presented above arise from guaranteeing CW operations *always* terminate despite the false rejection of operations by Byzantine storage-nodes. By relaxing this guarantee, the constraint on Q_C becomes:

$$Q_\mathrm{C}+b \le N-t,$$
$$Q_\mathrm{C} \le N-t-b. \tag{4.9}$$

Which leads to:

$$Q_\mathrm{C}+t+b \le N < 2Q_\mathrm{C}-t,$$
$$Q_\mathrm{C}+t+b < 2Q_\mathrm{C}-t,$$
$$2t+b < Q_\mathrm{C}. \tag{4.10}$$

So, the overall constraints on Q_C and N are:

$$2t+b+1 \le Q_C \le N-t-b;$$
$$Q_C+t+2b \le N < \mathtt{MIN}\left[2Q_C-t,\ \frac{3Q_C+b}{2}\right].$$

And the minimal constraints are:

$$Q_C = 2t+b+1;$$
$$N = 3t+2b+1.$$

The differences between the two sets of constraints translate to slightly different liveness properties of write operations that incur no write concurrency in the face of Byzantine storage-node faults. However, in both cases, safety is never compromised.

The first derivation (4.6) guarantees that, in the absence of write concurrency, writes will *always* complete. The second derivation (4.9) provides slightly weaker liveness guarantees than the first. It provides that in the absence of write concurrency and in the presence of Byzantine storage-node faults, writes *may* complete; it depends on whether responses from Byzantine storage-nodes are in the set of the $N-t$ responses collected by the client.

4.4.8 Liveness

If multiple CW operations are ongoing, storage-nodes may execute CW requests for distinct CW operations and thus prevent any CW operation from completing. In such a scenario, some form of “back off” and retry is required to allow forward progress. Techniques such as randomized exponential back off or some form of prioritized request queues at storage-nodes (if the storage-node “detects” contention) could work.

There is a trade-off between the liveness guarantee of the R/CW protocol and the minimum number of storage-nodes required. In practice, it is probably worth living with the reduced liveness properties in order to save $2b$ storage-nodes. This is especially true, since the power of a Byzantine storage-node can be mitigated through some synchrony assumptions (waiting for more responses—within some time bounds), or assumptions of a fair network (on which Byzantine entities cannot control message ordering). Under either assumption, it is likely that responses from a set of benign storage-nodes will eventually be collected, and that the operation will therefore complete. A more thorough discussion to the malicious rejection of CW operations is given in Section 4.6.3.

4.4.9 Safety

Read operations only return the value of the latest complete CW operation. They are linearized after the CW operation whose value they return.

CW operations only complete if they are conditioned on the latest complete CW operation. A complete CW operation is always observed as repairable or complete; if it is repairable, its value is written “forward” to a new timestamp preserving the conditioned-on version chain.

4.5 Protocol scalability

Consider a failure model with $t = b = 1$. The smallest configuration for this failure model is $N = 6$, $\texttt{COMPLETE} = 5$, and $\texttt{INCOMPLETE} = 3$. Larger configurations, which reduce the load on any given storage-node, are possible. For example, given the same failure model, another valid configuration is $N = 9$, $\texttt{COMPLETE} = 7$, and $\texttt{INCOMPLETE} = 5$. In the smallest configuration, each storage-node must execute requests for $\frac{5}{6}$ of the operations performed. In the larger configuration, each storage-node need only execute requests for $\frac{7}{9}$ of the operations performed. Thus, it is possible to add storage-nodes to the system to increase its throughput. If 3Δ storage-nodes are added to the system to improve the throughput, then the R/CW constraints (from Section 4.4.7), for $\Delta > 0$, become:

$$|Quorum| = Q_{\mathrm{C}} + b;$$
$$Q_{\mathrm{C}} = 2t + b + 2\Delta + 1 \quad (= \mathtt{INCOMPLETE} + t);$$
$$N = 3t + 2b + 3\Delta + 1 \quad (= Q_{\mathrm{C}} + t + b + \Delta).$$

The ability to increase system throughput in this manner is because the R/CW protocol is a threshold (or majority voting [Gifford 1979; Thomas 1979]) Byzantine quorum system [Malkhi and Reiter 1998a]. Since N is bound from above by $\frac{3Q_{\mathrm{C}}}{2}$, the greatest throughput that can be achieved via threshold quorums is $1.5\times$. The lower bound on the *load* of each storage-node is $\frac{2}{3}$. Intuitively, the load measure indicates the fraction of operations for which each storage-node must execute requests.

If other quorum construction techniques are employed (e.g., the M-Path construction [Malkhi et al. 2000]), then the lower bound on load is $\Omega(\sqrt{\frac{b}{N}})$. For the scale of the prototype metadata service, the use of threshold-quorums demonstrates the benefits of quorum techniques. The benefits of true quorum constructions, such as the M-Path construction, are only prominent once systems are very large.

4.6 Discussion

4.6.1 Erasure codes

Thus far the use of erasure codes within the R/CW protocol has been elided from the discussion. When using erasure coded data, the m parameter (used in decode) is constrained by the bound on `INCOMPLETE`; thus, $m <= Q_{\mathrm{C}} - t$. However, there are two main problems that arise from the use of erasure coded data.

First, additional mechanisms are required to ensure that Byzantine clients cannot perform poisonous writes (see Section 4.2.1). Detection of poisonous writes within the R/CW protocol hinges upon storage-nodes validating the data value associated with the CW operation with its verifier (the hash present in the timestamp). On the other hand, the client is responsible for the detection of poisonous writes in the R/W protocol (which utilizes erasure coding). As described in Section 3.2.3, the client detects poisonous writes through the regeneration of all N data-fragments, from which the cross checksum is recomputed and verified. For this approach to work in the R/CW protocol, the client would have to perform this validation on each value contained within the conditioned-on chain all the way back to the initial value (or until an agreed upon "correct" value, e.g., one that had been decided upon through garbage collection); this is impractical. One possibility is to use some type of verifiable sharing scheme (e.g., Verifiable Secret Sharing [Chor et al. 1985; Feldman 1987]), in which storage-nodes are able to validate the integrity of each CW operation; however, these schemes are currently very computationally and space inefficient.

The other limitation of using erasure coded data is that it requires the client to compute (and erasure code) the update before transmitting it to the storage-nodes. As will be discussed in Chapter 5, the R/CW protocol is extended to support arbitrary operations that are executed solely on the storage-node; e.g., if a CW object implements a directory, the client need only transmit the name it wishes to insert—the storage-node does the work of inserting the name into the directory. This approach implements replicated state-machines, as such it is not amenable to erasure coding.

Regardless of the limitations, erasure coding within the R/CW protocol may still be useful depending on the application's requirements. If the system model permits/prevents Byzantine clients from performing poisonous writes and if block storage is required, the R/CW protocol provides stronger semantics than does the R/W protocol.

4.6.2 Invalid authenticators

If the authenticator does not pass validation at a storage-node, the storage-node cannot tell if a Byzantine storage-node is involved, or if a Byzantine client corrupted a correct authenticator (or object history set). Digital signatures do not have this problem. As such, the storage-node can "reject" the CW request, and place the onus on the client to retry the R/CW operation using digital signatures. Other options include allowing the storage-node to reject the CW request outright (this gives a Byzantine storage-node the power to force a CW operation to abort) or allowing storage-nodes to perform a read operation to directly validate the object history set (this could require $O(n^2)$ messages in limited situations).

4.6.3 Storage-nodes rejecting CW requests

Byzantine storage-nodes may arbitrarily reject CW requests (based on the failure of object history set validation). As discussed in Section 4.4.8, a tradeoff in reducing the liveness of the protocol versus the constraints on N and Q_{C} exists. With the reduction in liveness Byzantine storage-nodes may be able to force clients to abort.

In addition to the ’solutions’ described previously (also in Section 4.4.8), i.e., the increased constraints on N and the assumption of a fair network, a third solution exists. This solution requires storage-nodes to provide sufficient evidence in the form of a valid object history set that supersedes the rejected CW request.

To provide this evidence, the storage-node must be modified in a number of ways. First, the replica history returned by storage-nodes in response to read requests must include the client ID of the client who issued the read request. The storage-node should generate the authenticator over the $\langle \mathit{ClientID}, \mathit{ReplicaHistory} \rangle$ tuple. Second, upon successfully executing a CW request, the storage-node should retain the object history set on which the CW is based. Third, in response to a rejected CW request the storage-node should reply with the set of “signed” $\langle \mathit{ClientID}, \mathit{ReplicaHistory} \rangle$ tuples as evidence that the rejected CW request is being rejected correctly.

Unfortunately, authenticators do not allow the client to directly validate the returned replica histories; signatures do not have this problem. If using authenticators, the client must validate the returned replica histories by sending the returned $\langle \mathit{ClientID}, \mathit{ReplicaHistory} \rangle$ tuple to the storage-node that originally authenticated the tuple. One additional modification must now be placed on storage-nodes; they must track the client IDs of all read operations (these logs can be purged similar to version pruning, as well this modification is not needed if using digital signatures). This modification is required for the storage-node to validate that the client ID present in the authenticated tuple is indeed correct. The use of the client ID ensures that a Byzantine storage-nodes cannot "reuse" replica histories, sent to them from correct clients, to arbitrarily reject requests (i.e., a client must have performed the read that resulted in the replica history). This prevents Byzantine storage-nodes from generating arbitrary, verifiable replica histories. Once enough replica histories have been validated the client can verify whether or not the request was rejected appropriately.

For example: with $N = 3t + 2b + 1, Qc = 2t + b + 1,$ then $\texttt{COMPLETE} = 2t + 2b + 1$ and $\texttt{INCOMPLETE} = t + b + 1$: For a storage-node to correctly accept a CW request at time T it would have to have observed an object history set with at least $\texttt{COMPLETE}$ logical timestamps of time T. Of those replica histories, validations from $\texttt{COMPLETE} - t$ can be awaited. Of the validations that occur, at most b may fail due to Byzantine storage-nodes, $\texttt{COMPLETE} - t - b$ pass validation. Since, $\texttt{COMPLETE} - t - b = t + b + 1 = \texttt{INCOMPLETE}$, Byzantine storage-nodes can make complete operations appear to be repairable. Rejecting a CW operation on the basis of hosting a repairable at a later timestamp is a valid action.

4.7 Evaluation

Since the R/CW protocol provides consistency more suitable to a metadata service than a block store, the majority of the evaluation is presented in the next chapter (where the R/CW protocol is extended for use in a metadata service). However, the R/CW protocol is not precluded from being used as a block based storage protocol. Thus, a brief evaluation of the protocol response time, when providing consistency for the storage of erasure coded data, is described below. Results pertaining to system throughput, concurrency, and scalability are presented in the Chapter 5.

The experimental setup is as follows. A rack of Intel P4 2.66 GHz machines with 1 GB of memory were used for the experiment. Each storage-node utilizes a dedicated 33.6 GB Seagate Cheetah 10K RPM SCSI disk. All nodes are connected through a single gigabit Ethernet switch.

Figure 4.11 shows the mean response time of read and write operations as the number

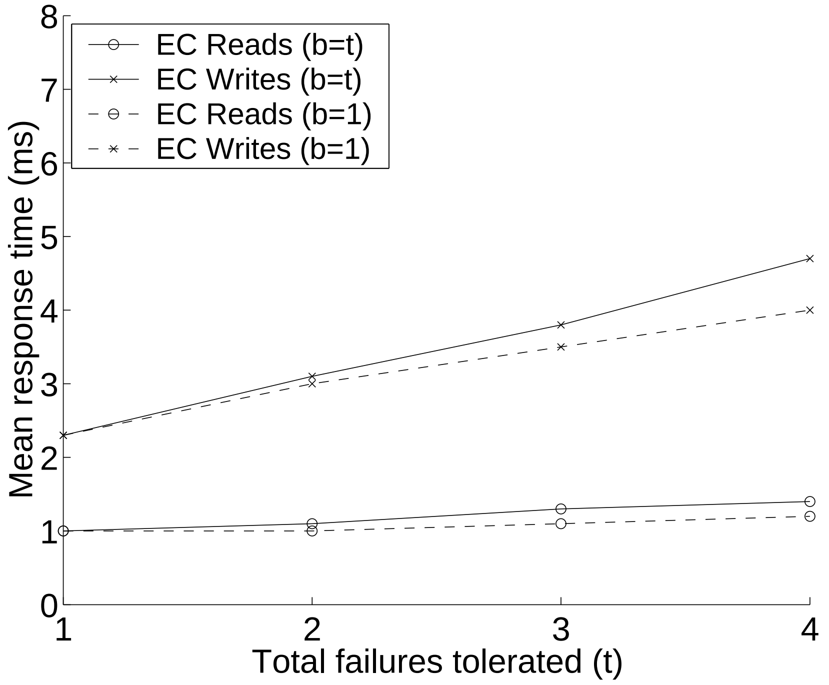


Figure 4.11: **Mean response time vs. total failures .** *This figure shows the mean response of the R/CW protocol when using erasure coded data. Separate lines are shown for reads vs. writes, as well plots for $b = t, N = 5t + 1$ and $b = 1, N = 3t + 3$, as t is scaled up, are shown. A block size, before erasure coding, of 16 KB was used.*

of tolerated failures (t) is increased. Separate plots for $b = t$ and $b = 1$, with $N = 3t + 2b + 1, m = t + b + 1$ are shown. Recall, the total storage blowup for erasure coded blocks is $\frac{N}{m}$. A block size of 16 KB, before encoding, was used for all data points.

As can be seen the slope of the `read` operation response time lines are very flat; this is similar to the slope of reads performed by the R/W protocol. This is because reads are space preserving; only 16 KB of data is ever transferred. Each operation is issued to all N storage-nodes, however, reads request only m data-fragments, the rest are read witnesses. As in the R/W protocol, read witnesses, can validate the returned data-fragments through the hash of the cross checksum stored within the timestamp.

As is expected, the slope of the `write` operation response times are steeper than those of the reads, since writes are not space preserving. As well, the difference in response time between the $b = t$ and the $b = 1$ write operation lines grow faster than it does for reads.

This is due to the rapid growth of N at $b = t$. As N increases, so does client computation time (to perform the encoding), as do communication costs. An increase in N also reduces network efficiency since smaller packets are being transmitted to a larger number of storage-nodes; recall, each data-fragment is of size $\frac{1}{m}$, so as N increases, so does m, thus data-fragment size is reduced.. The space-efficiency of both lines is almost the same; for example, with $b = t = 4$: $N = 21$ and $m = 9$ the total blowup is 37.3 KB, while with $b = 1, t = 4$: $N = 12$ and $m = 6$ the total blowup is 32 KB.

4.8 Summary

This chapter has developed a novel protocol that provides linearizability of R/CW operations. A conditional write operation performs a write to an object only if the value of the object has not changed since the object was last read. The R/CW protocol is useful for providing consistency of updates to metadata objects, although it can also be to provide consistency of block updates. The R/CW protocol provides read–modify–write semantics. It has been shown that many more powerful operations can be built with RMW semantics than with RW semantics (e.g., test-and-set). These operations are crucial to building fault-tolerant metadata services.

The R/CW protocol shares many features with the R/W protocol. It is designed around a hybrid fault model; it is extremely optimistic, optimized for low concurrency; and it is enabled by storage-node versioning. However, in order to fully tolerate Byzantine clients, replication must be employed. As well, the constraints on N and Q_{C} are higher. This chapter also showed that the R/CW protocol scales well as the number of faults tolerated is increased and when using erasure coded data. The next chapter extends the R/CW protocol into a query/update protocol and shows how a scalable metadata service and storage-system can be built.

5 Metadata Service

Scalability is a primary focus of many networked storage systems, including NASD [Gibson et al. 1998], Lustre [Braam 2004], and many recent SAN file system products. These systems all share a common design: a distinct metadata service managing a scalable collection of data storage servers. A similar high-level architecture is shared by recent research systems like Farsite [Adya et al. 2002] and Pond [Rhea et al. 2003], which logically separate metadata management from data storage.

For all of these systems, scalability and fault-tolerance of the metadata service are key challenges. The most common fault-tolerance solutions are agreement algorithms that perform state machine replication (e.g., using a protocol like [Bracha and Toueg 1985]). Unfortunately, such an approach does not scale as replicas are added. To make this approach scale, it is common to partition metadata across separate metadata servers (or replica sets). Unfortunately, unlike with data, this solution often comes with a visible change in semantics: loss of ability to perform atomic operations, such as rename, across directories stored on distinct metadata servers.

This chapter develops an alternate design for survivable, scalable, metadata services that maintains strong semantics. The architecture of this system is shown in Figure 5.1. Our *PASIS metadata service* (PMD) is "survivable" in that it relies on few assumptions about the environment in which it runs: it is designed to withstand arbitrary (Byzantine [Lamport et al. 1982]) failures of clients and a limited number of metadata-nodes, and requires no timing (synchrony) assumptions for correctness. In addition, it is "scalable" in that the addition of new metadata-nodes yields improvements in the capacity,

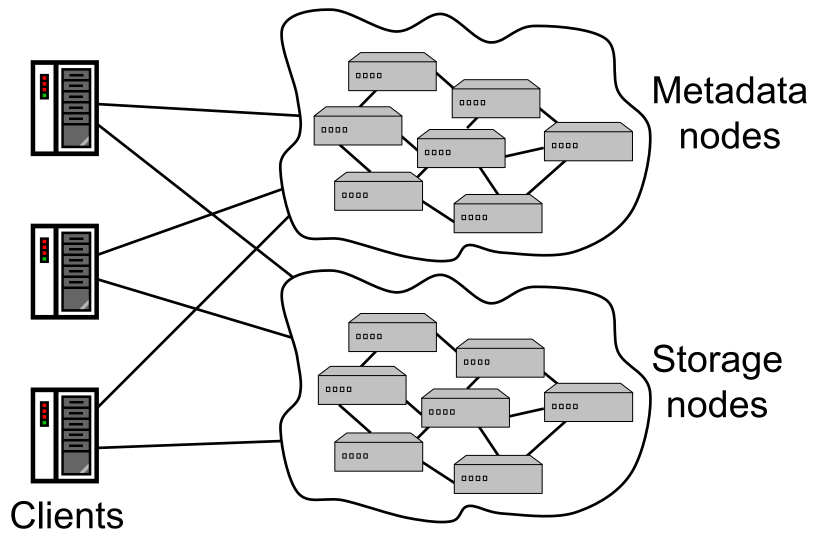


Figure 5.1: **Architecture of scalable storage systems.** *Traditionally, the focus of scalable storage systems has been scaling read–write storage to improve throughput, capacity, or fault-tolerance. The PASIS metadata service scales in a similar fashion: its throughput, capacity, or fault-tolerance can be improved by adding more metadata-nodes. Note that the metadata and storage processes can execute on the same hardware, even though the picture and most designs have them logically separated.*

throughput or fault-tolerance of the service; we refer to this as “horizontal scalability”.

The PASIS metadata service is constructed of metadata objects that utilize the R/CW protocol described in Chapter 4. While the R/CW protocol focused on reading and writing entire objects, this chapter extends the R/CW protocol to allow for more general *query* and *update* operations. These operations provide access to objects at a finer-granularity (e.g., reading/inserting directory entries vs. reading/writing full directories) (Sections 5.2.1 and 5.2.2). In addition, atomic updates across multiple objects are required: e.g., moving a file from one directory to another requires that the removal and insertion be performed atomically on the source and destination directories. Thus, a single conditional write operation that can modify multiple objects and can be conditioned on a superset of these objects being unchanged is introduced in Section 5.2.3. However, these extensions do not fundamentally alter how the R/CW protocol behaves. The bounds in terms of N and the thresholds remain the same, as does the optimistic nature of the protocol and its guarantees, all while avoiding expensive cryptography.

Moreover, by avoiding heavyweight agreement protocols, the PASIS metadata service offers horizontal scalability that has not yet been achieved for such a service. Our protocols derive from threshold voting protocols [Gifford 1979; Thomas 1979]. In such an approach, only subsets (i.e., a majority) of metadata-nodes need be accessed to complete an operation. As such, metadata-nodes can be added to improve capacity, throughput, or fault-tolerance. Moreover, the threshold voting approach employed can be extended to quorum systems that offer greater throughput scalability [Malkhi et al. 2000; Naor and Wool 1998].

5.1 Overview

Metadata objects are a type of R/CW object that provide metadata-specific interfaces. Read operations of R/CW objects are extended to be *query* operations of metadata objects; read operations read the R/W object, whereas query operations return the result of a deterministic read-only function performed on the metadata object. CW operations of R/CW objects are extended to be *update* operations of metadata objects; CW operations send an object replica in each request, whereas update operations invoke a deterministic function on the metadata object.

Since each metadata node performs the operation on its replica, metadata objects provide replicated state machine [Schneider 1990] semantics. These semantics prevent Byzantine clients from corrupting the state of metadata objects, since all updates are verified by the metadata servers. For example, metadata-nodes can prevent a Byzantine client from inserting an existing name into a directory object, because the metadata-nodes can only be manipulated by the appropriate operations (and the results can be verified by the metadata-node).

Two optimizations have been implemented to improve the efficiency of metadata objects. First, operations can be performed on metadata objects optimistically by sending only the operation and object history set to metadata-nodes; entire objects need not be transmitted across the network, thus reducing bandwidth. This is optimistic because, if the metadata-node does not host the candidate version on which the operation is to be performed, the metadata-node must re-sync its replica of the object (requiring an additional round-trip). Second, large metadata objects are broken into blocks, this aids in the reduction of bandwidth when syncing large replicas. When replica values must be fetched, only modified portions of the replica need be sent.

5.2 Metadata operations

To perform operations correctly, metadata-nodes must perform the operation on the version of the object replica that corresponds to the latest complete candidate. As in the R/CW protocol, the metadata-node requires the object history set to classify the complete candidate. As such, metadata operations build closely upon the R/CW protocol developed in the previous chapter. However, instead of shipping the data values (the results of client-side operations), the operations themselves are transmitted.

Metadata operations can be performed atomically on multiple objects. Since some operations span metadata objects, to provide failure atomicity it is necessary to perform these operations on multiple objects atomically. For example, `rename` removes a file from one directory object and adds it to another directory object.

This subsection describes two classes of metadata operations: *query* operations and *update* operations. As well, multi-object operations are discussed.

5.2.1 Query operations

Like read operations on R/CW objects, query operations on metadata objects are optimistic and complete in a single round in the common case. However, unlike read operations on R/CW objects, query operations do not return the contents of the entire object. This has a number of subtle implications. First, read witnesses (as described in Section 4.3.1) cannot be used for the results of query operations, since the value verifier (i.e., the object's hash) does not validate such results. Second, since the value verifier is com-

```
QUERY($Operation$) :
100: $\langle ObjectHistorySet, QueryResultSet \rangle$ := DO_QUERY($Operation$, QUERY_LATEST, $\perp$)
101: $\langle Candidate, Status \rangle$ := CLASSIFY($ObjectHistorySet$)
102: /* Iterate through returned object history set */
103: $CandidateResultSet := \emptyset$
104: for all ($M_i \in MetadataNodeSet$) do
105:     /* Add results from responses whose latest element matches the candidate */
106:     if MAX[$ObjectHistorySet[M_i]$] = $Candidate$ then
107:         $CandidateResultSet := CandidateResultSet \cup QueryResultSet[M_i]$
108:     end if
109: end for
110: /* Perform voting on the set of matching data results, need b+1 matching responses */
111: $\langle Count, Data \rangle$ := VOTE($CandidateResultSet$)
112: if ($Count < b+1$) then
113:     /* If less than b+1 results match, redo the query at the candidate's timestamp */
114:     $CandidateResultSet$ := DO_QUERY($Operation$, QUERY_LTIME, $Candidate.LT$)
115:     $\langle Count, Data \rangle$ := VOTE($CandidateResultSet$)
116: end if
117: /* If classification yields a complete candidate, return any of the matching votes */
118: if ($Status$ = CLASSIFIED_COMPLETE) then
119:     return (SUCCESS, $\langle Candidate.LT, Data \rangle$)
120: else
121:     /* $Status$ = CLASSIFIED_REPAIRABLE, perform repair */
122:     return (REPAIR_OPERATION($Operation$, $Candidate$, $Candidate.LT_{\text{conditioned}}$, $ObjectHistorySet$))
123: end if
```

Figure 5.2: **Client query pseudo-code.**

puted over the entire object, it can not be used to validate the data associated with a read response; instead, a voting scheme must be used.

The pseudo code for a read operation is shown in Figure 5.2. To perform a query operation, the metadata-node returns its replica history as well as the result of the query operation applied to the latest version of the object replica (cf. line 100). The client identifies the candidate by performing classification on the object history set, on line 101. Once the candidate is classified as complete, the client must determine the result of the query operation. Since only results pertaining to the latest timestamp in a replica's history are returned, the set of results corresponding to the candidate's timestamp must be constructed (cf. line 107).

The client then counts the votes in this set of results, see line 111. Matching results from $b+1$ metadata-nodes are sufficient "votes" for a client to use the result. Of course, more than $b+1$ object histories are required to identify the latest complete candidate. Since query operation results are returned optimistically based on the latest version hosted by the metadata-node, it is possible that no response attains a sufficient number of "votes".

If this is the case, the client performs the query metadata operation at a specific timestamp (the candidate's timestamp); see line 114. Finally, if the candidate is classified as complete, the result of the "voting" can be returned. Otherwise, repair is needed. Since the client does not hold a full copy of the object, repair is more complicated than described in the R/CW protocol and will be discussed later in the context of multi-object operations.

As an optimization if the query operation results are large, some metadata-nodes can act as witnesses by returning a hash of the operation's result. Voting can then be performed over the resultant set of hashes. Since the result of most query operations are small, the tradeoff between the computation time required to perform the hashing and the transmitting and comparison of the data value is in favor of the latter. (An exception may be the `readdir` operation as it returns a large number of directory entries).

5.2.2 Update operations

The CW operation of the R/CW protocol is extended for metadata objects to include update operations (e.g., `setattr` would update the attributes for an object). As in query operations, update operations do not transmit the object's new data value; only the operation to be applied to the replica and the object history set is sent. Allowing the metadata-nodes to perform update operations locally ensures the validity of the update. As in the R/CW protocol, updates are conditioned on the latest complete candidate, which is determined through classification of the object history set. Similar to query operations, client count "votes" on the results returned from the update operations.

If a metadata-node does not host the candidate, then it cannot safely perform the operation. In this case, the metadata-node must synchronize its object replica by fetching the state associated with the latest candidate. Object replica synchronization is discussed in Section 5.3.2.

Since the client does not have a local copy of the metadata object to update, it cannot construct the timestamp of the update operation (specifically, the verifiers in the timestamp). However, the timestamp can be constructed deterministically from the object history set and the resulting value of the updated metadata object. Since metadata-nodes have all of this information, they can be relied upon to deterministically construct the timestamp of the update operation (and it can be returned as part of the result). The calculated timestamp is then inserted into the replica's history.

5.2.3 Multi-object operations

Since all metadata object replicas are stored on the same set of metadata-nodes, the metadata-nodes can locally lock the set of object replicas being operated upon. Thus, a metadata-node can perform validation for each object replica accessed by the operation, and then, only if validation passes for all objects, execute the operation. This approach of validation has similarities to the validation phase performed in optimistic concurrency control [Kung and Robinson 1981]. However, one extra step is required in the validation of multi-object update operations. To prevent malicious clients from executing different operations across different objects at different metadata-nodes, the hash of the operation (including it's arguments and the set of objects the operation operates on) is included in the logical timestamp. This fixes the result of a multi-object update operation to a specific timestamp.

Multi-object operations complicate repair. Pseudo-code for the repair of multi-object operations is shown in Figure 5.3. If a repairable candidate is identified, then the client must request the operation that resulted in the repairable candidate (cf. line 200). Note that, since barrier-writes are always followed by an update operation, they need not be repaired. In response to the `READ_OPERATION` query, a metadata-node returns the operation and the object replica histories for all objects updated by the operation at the specified logical timestamp. Recall, the `QUERY` operation requires $b+1$ votes to return a result. The client constructs an object history set for each object updated by the operation by issuing a `READ_HISTORIES` query operation containing the object history sets of interest (cf. line 206).

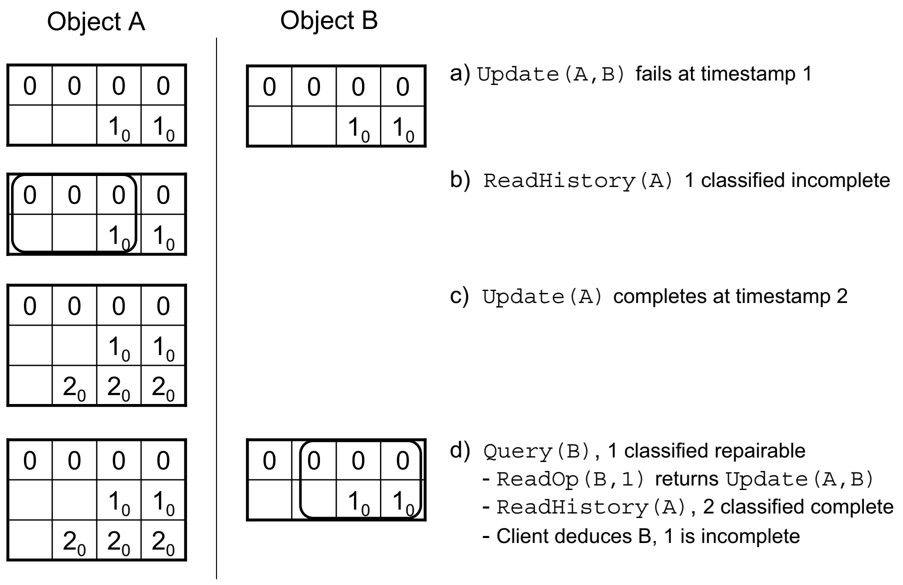
Next, a check is made to verify if a barrier-write across the sets of objects is required (cf. line 207). If a barrier is required then the client performs a barrier-write conditioned-on these object history sets (cf. line 208). If the barrier-write completes, the client re-

```
REPAIR_OPERATION($Object, LT$) :
200: $RepairOperation :=$ QUERY($\langle$READ_OPERATION, $Object, LT\rangle$)
201: /* Iterate through all objects in the returned operation */
202: $ObjectSet := \emptyset$
203: for all ($O_i \in RepairOperation$) do
204:    $ObjectSet := ObjectSet \cup O_i$
205: end for
206: $ObjectHistorySets :=$ QUERY($\langle$READ_HISTORIES, $ObjectSet\rangle$)
207: if (BARRIER_NEEDED($ObjectHistorySets$) = TRUE) then
208:    $\langle ObjectHistorySets, Status\rangle :=$ UPDATE($\langle$BARRIER_OPERATION, $ObjectSet\rangle$, $ObjectHistorySets$)
209:    if ($Status$ = FAIL) then
210:       return FAIL
211:    end if
212: end if
213: $\langle Candidate, Status\rangle :=$ REPAIR_NEEDED($ObjectHistorySets$)
214: if ($Status$ = FALSE) then
215:    return (FAIL)
216: end if
217: $Status :=$ UPDATE($RepairOperation, ObjectHistorySets$)
218: return ($Status$)
```

Figure 5.3: **Client multi-obj repair pseudo-code.**

constructs the object history sets and performs reclassification to ensure that repair is still required. A few cases exist in which repair is not required—the most obvious is when the operation being repaired has completed (or has been repaired by another client). More subtle cases are discussed a little later. If the client determines that repair is still required, an update operation corresponding the operation that is to be repaired is performed, line 217.

As mentioned repair may not be necessary for a few reasons. Usually, it will be the case that the histories returned from the barrier-write indicate that the *RepairOperation* has completed (or been repaired by another client). However, it is possible for some objects involved in a multi-object operation to be classified as repairable and others to be classified as incomplete (depending on the client's system view). If this is the case, it is not possible to repair all the candidates involved in the multi-object operation. The client can deduce, since some candidates involved in the multi-object operation are incomplete, that no candidate involved in the multi-object operation is complete (thus safely reclassifying the repairable candidate as incomplete). Likewise, by sending the set of object history sets for all objects updated by the multi-object operation to the metadata-nodes, the metadata-nodes can reach the same conclusion and allow such repairable candidates



Object A
Object B
a) Update(A,B) fails at timestamp 1
b) ReadHistory(A) 1 classified incomplete
c) Update(A) completes at timestamp 2
d) Query(B), 1 classified repairable
- ReadOp(B,1) returns Update(A,B)
- ReadHistory(A), 2 classified complete
- Client deduces B, 1 is incomplete

Figure 5.4: **Example of multi-object repair.** *For this setup:* $N = 4$, $\texttt{COMPLETE} = 3$, $\texttt{INCOMPLETE} = 2$. *(a) Initially an update operation is performed on Objects A and B. However, it fails part-way through. (b) A read history operation is issued to read the object history set associated with Object A. Logical timestamp 1 is classified as incomplete. (c) An update is performed, and completes, on Object A at logical time 2 (conditioned on timestamp 0). (d) A query operation is performed on Object B. Logical timestamp 1 is classified as repairable, thus repair must be performed. First, the operation resulting in the version at timestamp 1 must be read. It returns the original operation that updated Objects A and B. The history of Object A is then read and classified. Object A's timestamp 2 is classified as complete. From this, the client can deduce that Object B's version at time 1 could never have completed (otherwise Object A's version at 2 would have conditioned on time 1).*

to be over-written.

Similarly, it may be the case that some objects (in a multi-object operation) appear as complete, while others appear incomplete or repairable. Again, the client and metadata-nodes can come to the same conclusion by examining the object history sets of each object involved in the repair. Figure 5.4 shows an example of how multi-object repair works.

5.2.4 Summary

This section as described a number of extensions to the R/CW protocol that enables query and update operations to be performed against metadata objects. Instead of clients transmitting entire objects, query and update operations can be used to perform operations that

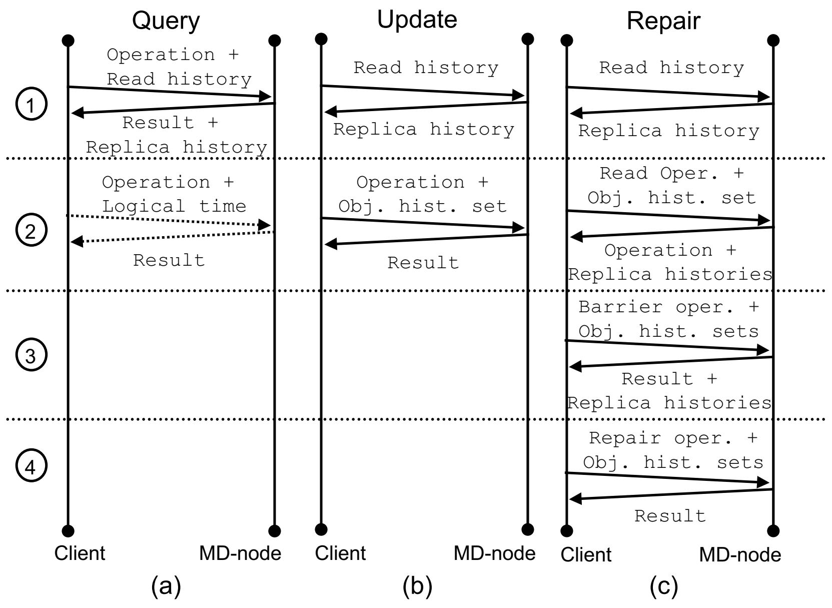


Figure 5.5: **Example metadata operations.** *Metadata objects support query and update operations. Both types of metadata operations may need to perform repair. The requests a client sends to metadata-nodes to perform metadata operations are shown.* **(a)** *Query operations may complete in a single round trip. If insufficient results are returned at the candidate's timestamp, a second request is sent to collect results at the candidate's timestamp.* **(b)** *For update operations, a client first requests replica histories to identify the candidate. Then, the client issues the update operation with an object history set constructed from replica histories returned by a recent query.* **(c)** *To perform repair a client requires the operation that resulted in the candidate. Since operations may span multiple objects, metadata-nodes potentially return many replica histories.*

are executed by each metadata-node. When metadata-nodes perform these operations they are able verify the integrity of the request and the result. Query operations require an object history set constructed from the replica histories returned by a recent query operation. To increase efficiency, object history sets can be cached by clients. Both query and update operations may require repair.

Figure 5.5 describes query, updates, and repair operations. Figure 5.5(a) illustrates the requests and replies a client exchanges with a metadata-node to perform a query operation. Figure 5.5(b) illustrates the requests and responses a client exchanges with a metadata-node to perform an update operation. Figure 5.5(c) illustrates the requests and responses a client exchanges with a metadata-node to perform repair.

5.3 Improving efficiency

Two additional considerations for metadata objects are presented within this section. The first is an optimization to handle large metadata objects. The second details an approach to efficiently synchronize object replicas.

5.3.1 Breaking large objects into blocks

Metadata objects could be very large (e.g., a directory object with thousands of files). To efficiently handle large metadata objects, metadata object replicas can be broken into fixed sized blocks. Even though the metadata object replica is broken into blocks, metadata operations still occur atomically on the metadata object.

If metadata objects can be stored in a structured fashion, update operations can be implemented to be considerate of block boundaries (e.g., not allowing directory entries to span blocks). In so doing, the number of blocks modified by an update operation can be minimized. If the state of metadata objects cannot be stored in a structured fashion, then techniques like those used in the Low Bandwidth File System [Muthitacharoen et al. 2001] could be employed to minimize the number of "chunks" that a metadata update operation modifies.

The value verifier of the logical timestamp for a CW operation on a large metadata object is a collision-resistant hash of the list of the replica's block hashes. The cost of generating the verifier is linear in the number of blocks that comprise the object. For extremely large objects, Merkle hash trees [Merkle 1987] should be considered.

5.3.2 Object synchronization

Since update operations only execute at a subset of metadata-nodes, it is possible for some metadata-nodes to become "out-of-sync" (i.e., to not host the most recent complete candidate). To perform an update operation, the metadata-node requires the version of the object at the candidate's timestamp. A metadata-node can "sync" its object replica by fetching the value corresponding to the latest complete candidate directly from another metadata-node. There is enough information in the object history set for the metadata-node to know which other metadata-nodes host the candidate. As well, the metadata-node can validate the correctness of the object value received with the verifier in the candidate's timestamp.

To sync a large metadata object, a metadata-node requests the hash list (or tree) for the candidate object from another metadata-node that hosts the candidate. The value verifier in the timestamp validates the correctness of the hash list returned. Given the hash list for the candidate object version, the metadata-node can request only the out-of-date blocks. The hashes in the hash list validate each block of the metadata object.

5.4 PASIS metadata objects

The PASIS metadata service (PMD service) exports a number of metadata objects. Each type of metadata object consists of internal state and provides a set of deterministic operations that can be performed on the object, as described in Section 5.3. Some operations span multiple objects—for example, a rename operation is performed on a pair of directory objects. Others may be read-only. This subsection describes the design of four types of metadata objects: directory objects, attribute objects, lock/lease objects, and authorization objects. Directory and attribute objects are fully implemented. Lock and authorization objects are designed but not yet implemented. Implementation details of directory and attribute objects are described within this section. The implementation of the PMD service, in the context of a distributed NFS framework, is described in Section 5.5.

The design of the PASIS metadata objects focuses on minimizing the access concurrency experienced by any one metadata object. Reducing the amount of update concurrency experienced by metadata objects improves the efficiency of the underlying R/CW protocol actions.

5.4.1 Attribute objects

An attribute object exists for each file stored in the PASIS storage service. The attribute object contains the per-file information expected by the clients (e.g., the NFS server and the NFS clients). In our implementation, these attributes map directly to typical UNIX file attributes (e.g., `mode`, `link_count`, `uid`, `gid`, `size`, `mtime`, `ctime`, etc.).

There is a tradeoff between storing attributes in separate objects versus storing them within their parent directory entry. If stored within the directory, operations that access attributes and directories need only access a single metadata object. However, storing attributes within directory entries increases the false sharing of the directory object for any operation that operates on the attributes without operating on the directory (e.g., `setattr` and `getattr`). Since there may be many files managed by each directory object, the read and update traffic for these attributes could generate frequent concurrent accesses to the directory object. Additionally, hard links (i.e., multiple names for the same object) cannot be easily supported if attributes are stored within directory entries.

5.4.2 Directory objects

Directory objects store the names and access information for files and other directories. Access information specifies how the named object can be accessed (e.g., where the objects are located and their encodings, not access control information). The access information for directories is PMD service specific. The access information for files is storage service specific. For example, if the R/W protocol is being used as the protocol underlying the storage-service, the access information will contain the protocol parameters (e.g., N, m, Q_C, etc.) and the encoding scheme being used (e.g., replication, IDA, etc.).

The attributes of a directory are stored in the directory object itself—a separate attribute object is not used. Since most operations that access a directory object also access the directory's attributes, this design decision does not contravene the design goal of separating objects to minimize access concurrency. Indeed, directories maintaining their own attribute information allows for greater efficiency at the storage-node and over the network: object histories need only be maintained (and returned) for the directory objects. On the other hand, since files can use a separate storage-service protocol, attributes and data must be updated independently.

The directory object, since it may be large, is stored as a collection of blocks (see Section 5.3.1). The directory object block size is 4 KB, in our implementation. A simple structure is used in the implementation of the directory objects; it is just a list of directory entries. Each directory entry is a $\langle$name, access$\rangle$ pair. The access information encodes the object's ID, the set of node IDs that host the named object, and the scheme which describes the encoding of the object (e.g., the replication factor). The object encoding is specific to the service owning the named object (i.e., either the PMD or the PASIS storage-service).

To look up a name in the directory object, a linear search of its directory entries is performed. When entries are added to the directory object, care is taken to avoid splitting them across block boundaries. When entries are deleted, no compaction is performed, but the free space created may be used for future entry insertions. Standard improvements to directory implementations, such as using b-trees to avoid linear searching, could be applied.

5.4.3 Lock (lease) objects

Lock objects provide serialization points. Locks are not needed for metadata object consistency, since the Q/U protocol ensures that all metadata operations occur atomically. However, lock objects may be desired by clients wishing to control access to data. As such the design and implementation of the lock objects is dependent on their use. By providing the ability to implement lock objects using the Q/U protocol, locks are guaranteed to have the same fault-tolerance, consistency, and scalability guarantees as the metadata.

If lock objects are implemented in this manner, the storage service must be able to validate locks presented to it. Recall, the storage service is implemented by a distinct set of storage-nodes. We envision three possible scenarios for lock validation. First, capabilities are generated as the result of lock operations; these capabilities can be verified by storage-nodes (as in NASD [Gibson et al. 1998]). Second, all accesses to the storage service are serialized through the metadata service. Third, each storage-node verifies each access before performing it; i.e., each storage-node issues a query operation to the PMD service that returns the lock status.

There are two basic uses of lock/lease objects in distributed file systems: to maintain client cache consistency within the storage service and to provide application locking of data (i.e., file locking). To maintain client cache consistency, clients must be notified of changes to cached data. In such an approach, callbacks from the metadata service would be needed to notify holders of cached data that the data is stale. To maintain the fault-tolerance of the system, the application server ought to wait for $b+1$ callbacks before acting; however, since caching is done for performance, not correctness, it is safe to invalidate cache entries based on a single callback.

Since fault-tolerant systems should not rely on potentially faulty clients to release locks, lock objects should provide lease semantics. Achieving lease semantics requires that locks timeout. The R/CW protocol is developed in an asynchronous model of time, so that invalid timing assumptions cannot break the properties provided by the R/CW protocol. In practice, loosely synchronized clocks are common and, if used wisely, can expire acquired locks.

5.4.4 Authorization objects

Authorization objects manage the privileges associated with metadata objects. There are two standard approaches to managing privileges: access control lists (ACLs) and capabilities. ACLs manage privileges on a per-object basis whereas capabilities manage privileges on a per-client/user basis. Either approach to privilege management can be implemented with authorization objects. An authorization object can be associated with each metadata object, and operations on the metadata object will only be performed if authorized.

Authorization objects may be needed for the storage service as well. Validation of authorization objects can occur similarly to the validation of locks. For example, the storage service can perform a read of the authorization object before permitting data to be read or

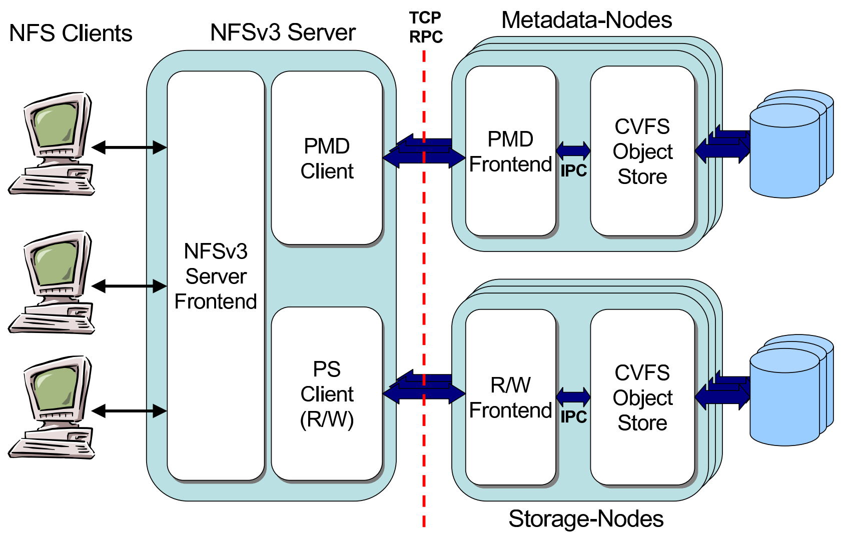


Figure 5.6: **PASIS storage system.** *Components of the PASIS storage system are shown above. The PASIS storage system is split into two components: a client and a set of storage-nodes. The client implements an NFSv3 server. The NFS server consists of a PASIS metadata (PMD) component and a PASIS storage (PS) component. A single NFS server is able to support multiple concurrent NFS clients. Alternatively, the NFS server may be mounted via loop-back on the same machine as the NFS client.*

written. Or, the application server can provide a capability to the storage service to read or write specific data.

5.5 Storage-system implementation

This section describes the metadata objects and storage service that comprise the prototype file system. For the storage service in the prototype file system, we use the PASIS read–write protocol from Chapter 3. The PASIS storage service (*PS service*) runs on storage-nodes (similar in nature to the metadata-nodes the PMD service runs on). An example configuration is shown in Figure 5.6. Each NFS server interacts with metadata objects implemented by the PMD service, as well as data objects stored within the storage service. Many distinct NFS servers exporting the same file system image can operate concurrently against the PMD service and PS service.

Operation	**Type**	**Objects**	**Description**
`readhist`	Query	Attr. or Dir.	Read object history
`create`	Update	Attr. & Dir.	Create a file
`remove`	Update	Attr. & Dir.	Remove a file
`mkdir`	Update	Dir. & Dir.	Create a directory
`rmdir`	Update	Dir. & Dir.	Remove a directory
`symlink`	Update	Dir. & Attr.	Create a symbolic link
`readlink`	Query	Directory	Read a symbolic link
`getattr`	Query	Attribute	Read file attributes
`setattr`	Update	Attribute	Write file attributes
`lookup`	Query	Directory	Read file's access info.
`readdir`	Query	Directory	Read entire directory
`rename`	Update	2 Dir. & 2 Attr. or 3 Dir.	Move a file/directory
`link`	Update	Dir. & Attr.	Create a hard link

Table 5.1: **Implemented PMD service operations.**

5.5.1 Metadata operations

Table 5.1 lists the set of metadata operations that are currently implemented by the PMD service. The operations are inspired by NFS, but are generic enough to support many file system instances. The **Type** field specifies whether the operation is an update or query operation. Example query operations include: `getattr`, `lookup`, `readdir`, and `readlink`. The **Object** field specifies the number and types of metadata objects on which that operation operates. In the case of operations that span multiple objects, more than one metadata object is listed. For example, `remove` modifies the parent directory object and the `link_count` attribute stored within the file's attribute object.

As can be seen, many operations operate on directory objects. Many of these operations modify directory attributes as well as modifying directory entries (e.g., `create`, `remove`, etc.), thus justifying our design decision to encapsulate attributes within the directory object.

5.5.2 PMD metadata-nodes

The metadata-nodes use the Comprehensive Versioning File System (CVFS) [Soules et al. 2003] to store data objects and their versions. The query/update extensions to the R/CW protocol, as described in Section 5.3, have been implemented, as have object synchronization and multi-object repair. Additionally, each metadata operation described in Table 5.1

has been fully implemented.

CVFS objects

On the metadata-node, each metadata object is associated with three CVFS objects. One CVFS object is used to store the metadata object's internal state (e.g., the directory structure). Attributes are stored within the extended attribute field of this CVFS object's attributes. Another CVFS object stores the metadata object's history, while the third stores a hash tree computed over the object's internal state (to support large objects, see Section 5.3.1). The metadata object's history and internal state are versioned on every update. These versions can be garbage collected once the metadata-node classifies a later update operation as complete (i.e., on the next successful update of the metadata object). Note, completed barrier operations do not result in this version history compaction.

Object histories

Along with the metadata object's history, query operations optimistically return the result of the operation performed on the latest version of the metadata's internal state (as described in Section 5.2.1). A special query operation,`readhist`, is used to read only an object's history. Batching of `readhist` results is supported (i.e., history from multiple objects can be returned by a single call). As well, all update operations also return the history associated with each object present in the operation. This history can be cached by clients to reduce the number of read history queries. Each metadata-node generates N authenticators over the object histories using HMACs based on pair-wise symmetric keys. We use a publicly available implementation of MD5 for all hashes [Rivest 1992]. Each HMAC is 16 bytes long.

Object locking

Upon receiving an update operation, the metadata-nodes locally lock each object replica accessed by the operation. When locking an object's replica, care is taken to preserve operation ordering at that storage-node. This can help prevent unnecessary object syncing from occurring when objects are executed out-of-order, as is described in the following example.

Imagine the following sequence of operations pending at a single metadata-node at the same time: 1) `create` (a, /), 2) `create` (b, /), 3) `setattr` (b). It should be noted, that, if a correct client performed the operations, it is only possible for operations (2) and (3) to be pending concurrently if operation (2) has completed successfully and operation (3) is conditioned on (2); this can occur on a slow storage-node, since only a subset of the updates need to complete for the operation to complete, but updates are transmitted everywhere. If only object locking is performed without preserving operation ordering: operation (1) locks the '/' directory; operation (2) blocks on the lock held by the '/' directory; operation (3) attempts the `setattr` although the `create` has not yet completed on this metadata-node—in this case object syncing would attempt the create.

Update operation validation

After each replica within the operation has been locked, each object history set is validated. Once validation has successfully completed (for all objects), the update operation is performed. Validation is the same as for the R/CW protocol, with two exceptions. First, since the conditioned-on timestamp is calculated from the object history set (passed in by the client), no validation is performed on the condition-on timestamp (line 735 and 741 of Figure 4.8). Second, since update operations are transmitted, as opposed to full data objects in the R/CW protocol, there is no *Verifier_Data* to validate. However, if repair is being performed, metadata-nodes must validate that the correct operation is being performed. To do this the operation hash is compared to the repairable candidate's operation hash; recall, the operation hash is stored within the timestamp.

If the operation completes successfully, a hash is generated over the replica's updated contents and is added to the object's hash tree. Each replica history is updated with the new timestamp computed from a hash of the object's hash tree, the operation's hash, and the hash of the object history set (which was used to validate the operation—as described in Section 5.2.2).

Object name uniqueness

Each object within the PMD service is given a unique object ID (OID). Likewise, each file stored by the storage service is also identified by an OID. Object IDs are stored within directory entries to uniquely identify the file or directory to which the entry is linked. Within the PASIS storage system, OIDs are similar to the inode numbers used by traditional file systems (or filehandles used by NFS). However, unlike traditional file systems, OIDs are not be centrally assigned. This complicates the validation performed during object creation.

In the PASIS storage system, the client is responsible for generating a 256 bit OID. The client generates a 256 bit random number that it uses as the OID. The client then performs a read history query operation on the newly generated OID. If a metadata-node hosts the OID, it returns the replica history associated with the OID, if not, the metadata-node returns a special *null replica history* (a history with a single timestamp of 0). As well, the history of the parent directory object is also read.

When performing a `create` or a `mkdir` operation, the metadata-node validates the object history set to ensure that the create OID's latest complete timestamp is 0. If a create operation succeeds (i.e., it receives successful responses from $Q_{\mathrm{C}} + b$ metadata-nodes), the client is ensured that the OID it generated is globally unique. If a create operation fails (i.e., is classifiable as incomplete), the metadata-node is free to accept a create operation from a different client of the same OID; since the latest complete timestamp is still 0. The null history entry remains part of the replica's history until it is pruned by a subsequent update operation that observes a completed create. Validation is similar for the repair of a create operation: 0 must be the latest complete timestamp; and the operation hash of the repair operation must match the operation hash stored within the repairable candidate's timestamp.

To remove an OID (e.g., through a `unlink` or a `rmdir` operation), the replica history associated with the OID must be reset to the initial null value. Thus, the OID is only free once a remove operation has completed successfully.

5.5.3 PMD clients

A client library has been implemented to facilitate interfacing with the PMD service. The library's interface consists of the set of metadata operation service calls (with the exception of `readhist`, which is not exported externally). The implementation of the query and update operations follows the presentation in Section 5.3.

NFS server

A NFSv3 server has been implemented that uses the client library. All NFS metadata operations have been mapped to PMD service operations. NFS data operations (file read/write) are mapped to calls within the storage-service. There is a one-to-one mapping between NFS filehandles and PASIS OIDs.

Some NFS operations require multiple PMD operations. For example, there is a disconnect between the arguments of the NFS `unlink` operation and the PMD `unlink` operation. The NFS `unlink` operation takes a filename and a directory file handle as arguments, while the PMD `unlink` operation requires an additional argument, the filename's OID. The filename's OID maps to the attributes of the file, which may be updated by the unlink (e.g., the link count would be decremented). In order to perform this update operation, validation must be performed over the object's history set. Thus, the OID of the filename's attributes is required to construct its object history set. Therefore, a PMD `lookup` is performed prior to the unlink operation. Additionally, during the PMD unlink operation, the metadata-node validates that the filename matches the OID passed in.

Client history caching

To reduce the number of read history operations, object history sets are cached by the client. Every metadata operation request in the PMD service returns a replica history from each metadata-node executing the request. Histories are returned even if the request fails to execute. Since histories are cached, they may become out-of-date, or *stale*. A stale replica history will cause the request to fail validation at the metadata-node from which the replica history originated (see line 728 in Figure 4.8). An up-to-date replica history is returned by the metadata in response to receiving a stale history; thus, the client can update its cache and retry the operation.

Retry and concurrency

Although the NFS server locks each filehandle associated with each operation at the PMD client, operations may still abort due to concurrency. Thus, operation retry is necessary. Upon retry, new object histories must be obtained and classified. The operation is based upon these new histories. Many different policies regarding backoff and retry may be implemented to avoid retrying operations concurrently. This is particularly relevant in the face of repair, since repairs issued concurrently may cause livelock if they execute at metadata-nodes in an interleaved order that prevents any repair from completing successfully. This work does not focus on the policies regarding backoff and retry, however it is discussed further in the evaluation section.

5.5.4 Storage service

The PMD service is one part of a complete system, the storage service and the application server complete the system. In the case of a file server application, there is much flexibility in how the metadata objects are used to provide file services. For example, locks, access privileges, and client caching of stored files involve the PMD service and the storage service. This subsection briefly describes the selection of a storage service.

The interface and access protocol used by the storage service is independent of the protocol that underlies the metadata service. For example, the storage service may use either a block based (e.g., iSCSI or Fibre Channel) or an object based protocol to access storage. However, some coordination is required in the design of the metadata objects and the interfaces provided by the storage service. For example, if objects (i.e., files in a flat namespace rather than blocks) are exported by the storage service, the metadata objects need not implement inodes or other structures to track block allocations.

Although we reject partitioning the namespace as a means to scale the metadata service, partitioning data for the storage service is reasonable. A storage system need not provide any guarantees about operations performed across multiple data objects; as such, partitioning is an appropriate technique for stored data. Partitioning allows different files to have different performance and reliability properties (e.g., `/tmp` need not be highly replicated).

Implementation

The PASIS read–write protocol, described in Chapter 3, underlies our storage service. It provides block granularity read/write access to objects. The R/W protocol provides strong consistency (linearizability of block read/write operations) and fault-tolerance of erasure coded data (e.g., data encoded with Rabin's information dispersal [Rabin 1989]). A PS service, implemented using the R/W protocol, can be relied upon to serialize all accesses to stored data. Such an approach is suitable for an application that controls concurrency itself. Alternately, locks could be provided by the PMD service so that the application does not need to provide concurrency control. Our PS service implementation also uses CVFS as its backing store; storage-nodes can either run collocated with metadata-nodes or not.

Another option is to use the R/CW protocol. The R/CW protocol offers stronger consistency semantics in that writes not based on the most current version will be rejected. This has the nice property that the application server can implement very weak cache consistency (since writes based on stale reads will be rejected by the storage service). However, these semantics come at an increased cost in terms of N and space-efficiency. Instead, the versioning capability of the R/W protocol could be used to provide strong consistency across the entire file system through the use of immutable files. Clients that write files through the PS service could be required to update the associated metadata attribute object with the version of the latest completed write operation; thus ensuring consistency between the data and the attributes. However, this requires all data transfers to be serialized through the metadata.

5.6 Evaluation

5.6.1 Experimental setup

All experiments are performed on a rack of 30 Intel P4 2.66 GHz machines with 1 GB of memory. Each computer has two 33.6G Seagate Cheetah 10K RPM SCSI disk drives and an Intel Gb Ethernet NIC. The computers are connected with a 24-port Gb switch. Debian testing Linux kernel 2.4.22 is installed.

Many experiments use NFS servers as clients to the PMD service, while others communicate directly to through the PMD library interface. Multiple NFS servers are able to access the same PMD service simultaneously. The NFS servers are mounted via loopback on the same machines as the NFS client. The NFS servers implement the NFSv3 protocol. The NFS servers use buffer cache of 128 MB. Unless otherwise specified, the buffer cache is write-through and data is expired after 10 seconds. No attributes or metadata is cached by the NFS servers.

The storage-nodes use CVFS as the backing data store. Each storage-node has a frontend that communicates with CVFS over IPC. Each CVFS instance uses a 512 MB buffer cache. All experiments show results using write-back caching at the storage nodes, mimicking availability of 16 MB of non-volatile RAM. This allows us to focus experiments on the overheads introduced by the protocol and not those introduced by the disk subsystem.

5.6.2 Cryptography performance

Authenticators are HMACs, based on MD5 hashes, taken over object histories. In the common case, the object history will have one or two entries. An object history with two timestamp entries is 112 bytes in size. It takes 1.33 μs to generate a single entry in the authenticator vector. For a very large object history, with 32 entries, a single entry in the authenticator vector takes 11 μs to generate.

Directory objects are implemented with 4 KB blocks (the large objects optimization). On update operations, MD5 hashes of modified blocks are taken. Such hashes take 24.4 μs to generate over a full block. However, the hash is only taken over the utilized portion of the block.

5.6.3 PMD micro-benchmarks

This subsection describes a number of micro-benchmarks performed against the PMD service. No file data is involved for any of these experiments, they only test the PMD service. The first set of experiments examine NFS micro-benchmarks. The second set examines the impact of concurrency on PMD `create` operations and the third set examines the impact of a fault on the response time distribution of a run of `create` operations.

PMD NFS micro-benchmarks

All the PMD operations described in Table 5.1 have been implemented. Most of the NFSv3 operations map to corresponding PMD service operations, although a few require multiple PMD operations. NFS micro-benchmarks were performed against some of the NFS metadata operations. The mean response times for these operations are listed in Table 5.2. The PMD service was configured to tolerate one Byzantine fault; therefore six storage nodes were used. In addition to the end-to-end response time for the PMD service, the response times as observed by the PMD storage-nodes are also listed.

The response time for the `create` operation represents a create that occurs within a directory comprised of a single block. Due to the implementation of directory objects, a linear search is performed to ensure the name being inserted into the directory does not exist; thus, the larger the directory is in size, the longer the create takes. Likewise, the performance of the `readdir` operation is also dependent upon the size of the directory (since each directory block is being transmitted back to the client). Thus, two results for `readdir` are shown: one for a small directory containing 5 entries and one for a large directory containing 500 entries.

Operation	**PMD end-to-end (ms)**	**PMD time on S-N (ms)**
`create` (in a single block)	1.73	1.00
`getattr`	0.34	0.02
`link`	1.05	0.71
`lookup`	0.74	0.26
`readdir` (small)	0.79	0.06
`readdir` (large)	1.38	0.09
`remove`	1.48	0.60
`rename`	2.83	1.19
`setattr`	0.58	0.22
`readhist`	0.72	0.09

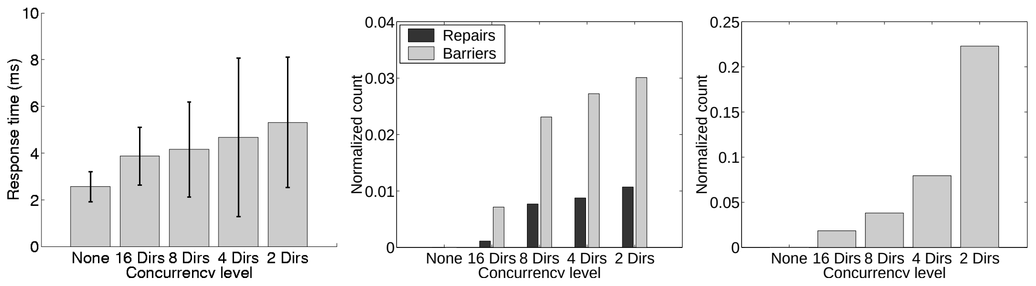
Table 5.2: **Micro-benchmarks of NFS operations.**

Concurrency

The impact of concurrency is examined in the context of PMD `create` operations. Three graphs show the results of performing PMD `create` operations with varying degrees of concurrency with $t = 1, b = 0, N = 4$. Two clients simultaneously perform create operations within a set of shared directories. Recall, `create` is a multi-object operation. In this experimental setup, the parent directory is the source of concurrency. To increase the likelihood of concurrency the set of shared directories is decreased between each run. In each run, each client randomly picks a directory to use as the parent by the `create` operation. Each client has only one outstanding request at a time. Care was taken to fully overlap the execution of both clients.

The first graph, Figure 5.7(a), shows the mean response time and standard deviation as concurrency is increased. The “None” bar represents a run with sixteen directories and only one client (i.e., there is no concurrency). It is not surprising that as the amount of concurrency increases, so does the response time as does the standard deviation. As concurrency is increased, repair and barrier operations become more common, as do the number of operations that must be retried due to stale object histories. Even at high concurrency levels (two clients sharing two directories), the mean response time and standard deviations are within a factor of two or three of a run with no concurrency.

The second graph, Figure 5.7(b), shows the total number of barrier and repair operations attempted at each concurrency level. These counts are normalized to the total number of create operations performed. Again, as concurrency is increased, the number



(a) Mean response time. **(b) Barriers and repairs.** **(c) Stale histories.**

Figure 5.7: **Concurrency experiments.** *These three graphs show the results of performing `create` operations with varying degrees of concurrency. Two clients simultaneously perform create operations into a fixed set of directories. For each operation each client randomly picks a parent directory from the directory set. To increase the degree of concurrency the directory set size is decreased between each run. (a) Shows the mean response time and its standard deviation as concurrency is increased. (b) Shows the total number of barrier and repair operations performed at each concurrency level normalized to the number of create operations issued by the client. (c) Shows the total number of times an operation was rejected due to stale object histories, again normalized to the number of create operations.*

of barrier and repair operations also increase. It is interesting to note that there is a large step increase from the very low concurrency configurations (“None” and “16”) to the higher concurrency levels (“4” and “2”).

The third graph, Figure 5.7(c), shows the number of total operations (including repairs and barriers) that are rejected due to stale object histories. Recall, the client caches replica histories returned by recent operations to construct the object history set for a subsequent update operation. As well, every operation returns an updated replica history (even if that operation failed). Examining the steep increase in stale object histories at the very high concurrency levels, one notices that there is often a race between the two clients trying to repair the same parent directory. Both clients try to write barriers, but neither client quite succeeds in completing a barrier (since the barrier is rejected from a subset of the nodes because the other client just wrote its barrier there). This observation requires the designer to carefully consider the back-off and repair policies when using optimistic protocols that can result in livelock. More work is required in this area.

Tolerating faults

This experiment shows the impact of a fault occurring at a random point during a run of `create` operations. The system was configured with $t = 1$, $b = 0$, $N = 4$ and a single client that performed unique `create` operations continuously into one of sixteen directories (picked at random). The client had only a single request outstanding, so there is no concurrency. Each run lasted for 10 seconds. Randomly during each fault-induced run, one of the storage-nodes was killed. Seven fault-induced runs were performed. There were no correctness problems present in any of the fault-induced runs. The remainder of this subsection quantifies the performance consequences of running with a failed server.

Figure 5.8 shows the mean response time and standard deviation of a single fault-free run and the accumulation of the response times from the seven fault-induced runs. The mean response time for the fault-free run is 2.17ms with a standard deviation of 0.34ms. The mean response time across all fault-induced runs is 2.47ms with a standard deviation of 0.35ms. Although the standard deviations of the fault-free and the set of fault-induced runs are similar, the standard deviation of each individual fault-induced run was between 0.52ms and 0.66ms. In general, fault-induced runs with a higher mean response time also had a higher standard deviation. Runs with higher mean response times also generally have a larger number of outlier response times (response times > 3ms).

In a fault-free run, the client only waits for the fastest $N - t$ responses. Once a failure has occured, the client still waits for $N - t$ responses, but there are now only $N - t$ servers, so it is waiting for all responses (rather than the fastest subsets). Thus, variations in individual storage-node response times are not masked. This accounts for the observed increases in the averages and standard deviations of response times within the fault-induced runs.

5.6.4 PMD service macro-benchmarks

We use the Postmark benchmark [Katcher 1997] to benchmark the performance of the PMD service. Postmark is a metadata intensive benchmark and provides useful infor-

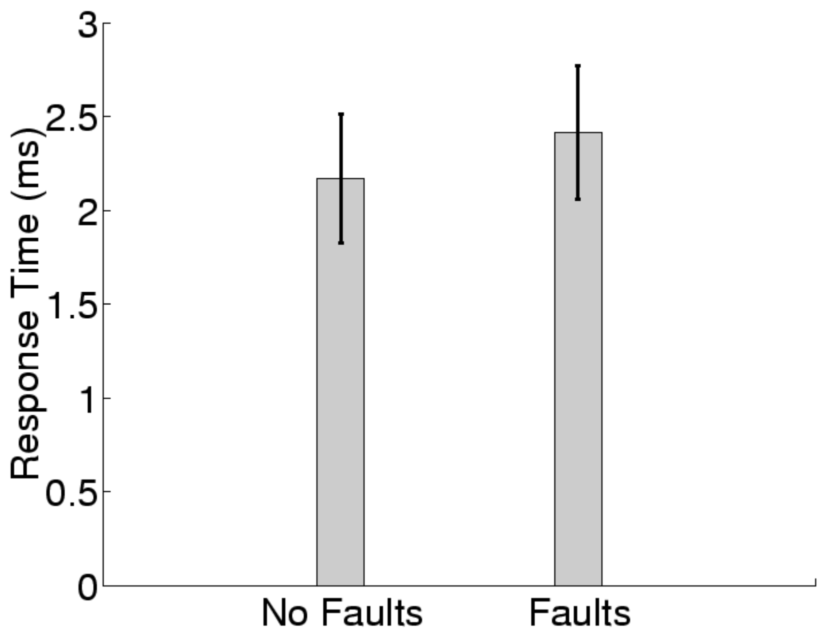


Figure 5.8: **Response time distributions of a fault-free and multiple fault-induced runs.** *This figure shows the mean and standard deviation of response times for a fault-free run and multiple fault-induced runs for the* `create` *operation. At a random point during each of the other runs, the PMD process on one of the storage-nodes is killed. As can be seen, the mean response time for the fault-induced runs is higher than in the fault-free run. Although the standard deviations are almost the same (between the fault-free and the all fault-induced runs), the standard deviation in a single fault-induced run is higher than the standard deviation in the fault-free run.*

mation about the performance of the PMD service. Postmark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services. Postmark is comprised of two phases: (i) in the *creation* phase, it creates a large number of small randomly-sized files (between 512 B and 9 KB); and, (ii) in the *transaction* phase, it performs a specified number of transactions. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. Three Postmark configuration parameters are important to our experiment: *files* (determines the creation phase), *transactions* (determines the transaction phase), and *directories* (determines degree of access contention). Results from Postmark experiments are given in transactions per second over both phases.

PMD service base throughput

The first experiment determines the maximum throughput of the PMD service. The PMD service is configured to tolerate a single benign metadata-node fault and Byzantine clients: $t = 1$, $b = 0$, and $N = 4$. Postmark is run on an increasing number of NFS servers to determine the maximum throughput of the PMD service in this configuration. To ensure that the PMD service is as loaded as possible, the NFS server uses local storage for the storage service. Postmark is designed to benchmark a single NFS server. However, it is being used to benchmark a decentralized service behind an NFS interface. As such, we "scale" the number of files, transactions, and directories for each Postmark/NFS server down, as we scale the number of Postmarks/NFS servers up. This is done to maintain a consistent working set across runs. Each NFS server runs Postmark in a different directory of the PMD service. The working set fits within the cache on the metadata-nodes.

Figure 5.9 shows the throughput of the PMD service with up to 16 distinct NFS servers. For a single NFS server, Postmark is configured for 32768 transactions, 1024 files, and 64 directories. Each NFS server has a single Postmark benchmark run against it. The Postmark configuration is scaled down as the number of clients is scaled up, thus keeping the working set size the same. For example, with 16 NFS servers, Postmark scales down to 2048 transactions, 64 files, and 4 directories. The PMD service saturates just below 350 transactions per second.

Scaling fault-tolerance

In this experiment we evaluate the impact on throughput of adding metadata-nodes to scale the fault-tolerance of the PMD service. A single NFS server running postmark with a configuration of 4096 transactions, 128 files, and 1 directory generates load (note this configuration for Postmark differs from the above experiment).

This experiment is performed with two configurations: a *benign* configuration in which the number of crash recovery failures tolerated is scaled from $t = 1$ to $t = 3$, while $b = 0$; and, a *Byzantine* configuration in which the number of Byzantine failures

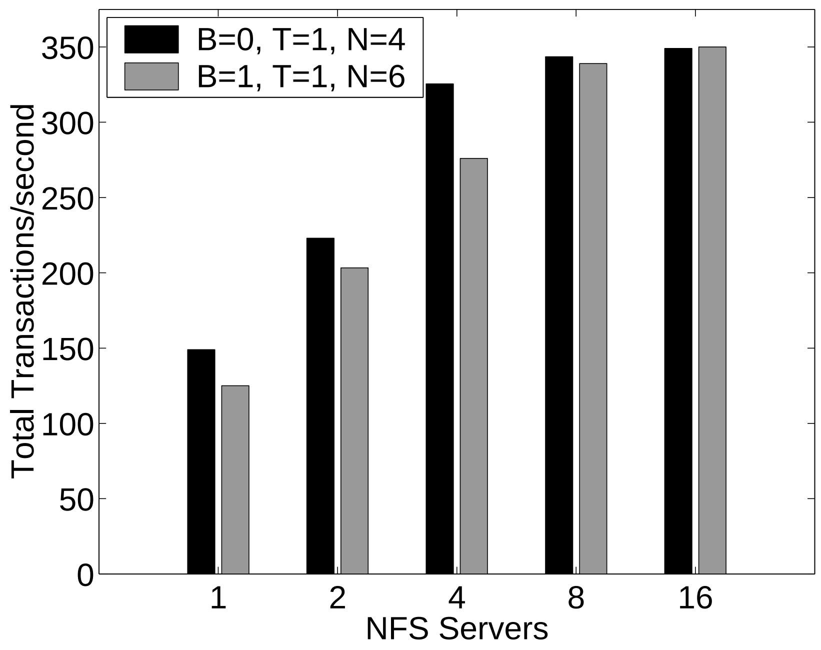


Figure 5.9: **Postmark throughput vs. client load.** *This graph compares the total system throughput of a Postmark workload as the number of NFS servers (PMD clients) increase in fault-free operation. Each client runs a single instance of Postmark against a NFS server mounted via loop-back on the same machine. The sets of bars represent a configuration with* $b = 0, t = 1, N = 4$ *and* $b = 1, t = 1, N = 6$.

tolerated is scaled from $b = t = 1$ to $b = t = 3$. Figure 5.10 demonstrates that as the number of failures tolerated scales, the responsiveness of the PMD service is fairly flat for the benign configuration and degrades only moderately for the Byzantine configuration. This degradation is expected, for a number of reasons. First, more cryptography is being performed by metadata-nodes (e.g., for $b = 3$ authenticators are comprised of 16 entries, since $N = 5b + 1$). Second, all storage-nodes are being communicated with, thus the communication costs grow as N increases. Additionally, this experiment shows the performance cost of a fully Byzantine-tolerant system is not prohibitive (at least for low values of b).

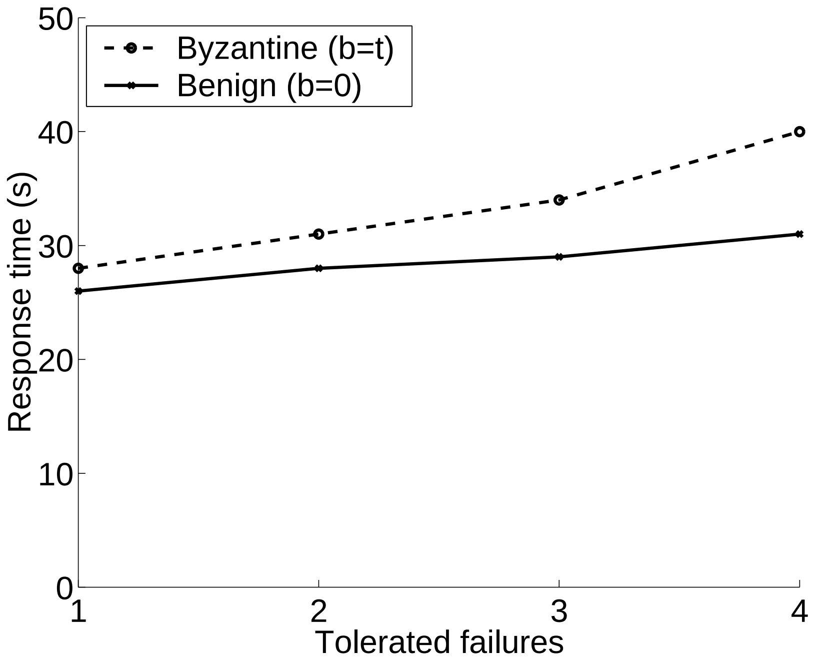


Figure 5.10: **Postmark run time vs. total failures tolerated (t).** *This graph compares the runtime of a Postmark workload as the number of tolerated faults (t) is scaled upward. The two lines represent a wholly crash environment with* $b = 0$*, while the other represents a wholly Byzantine environment with* $b = t$*.*

Scaling throughput using threshold quorums

Recall, Section 4.5 describes techniques that can be used to scale the system's throughput by adding storage-nodes. For example, the smallest configuration with $t = 1, b = 1$ is $N = 6$, $\texttt{COMPLETE} = 5$, and $\texttt{INCOMPLETE} = 3$. It is possible to increase the bounds on the R/CW constraints in the following way: by adding 3Δ to N, 2Δ must be added to Q_C, $\texttt{COMPLETE}$, and $\texttt{INCOMPLETE}$. Thus, as Δ increases the lower bound on the load of the system is $\frac{2}{3}$.

This experiment validates this hypothesis for $\Delta = 0$ to $\Delta = 7$. Table 5.3 shows the experimental setup for the threshold quorum based experiment. The table's first column shows Δ. The second column shows the value of N corresponding to that Δ value (for $t = 1, b = 1, N = 6 + 3\Delta$). The third column shows the size of each threshold quorum. The threshold quorum represents the set of storage-nodes a single client will communicate

Δ	**N**	**Threshold quorum size**	**Percentage of reqs each storage-node executes**	**Normalized system throughput (to $\Delta = 0$**
0	6	5	$\frac{5}{6} = 83.3\%$	1
1	9	7	$\frac{7}{9} = 77.8\%$	1.07
2	12	9	$\frac{9}{12} = 75\%$	1.11
3	15	11	$\frac{11}{15} = 73.3\%$	1.14
4	18	13	$\frac{13}{18} = 72.2\%$	1.15
5	21	15	$\frac{9}{14} = 71.4\%$	1.17
6	24	17	$\frac{17}{24} = 70.8\%$	1.18
7	27	19	$\frac{19}{27} = 70.4\%$	1.18

Table 5.3: **Threshold quorum experiment parameters ($b = t = 1$).** *This table shows the derived parameters when using theshold quorums. The first column shows Δ. The second column shows the value of N. The third column shows the size of each threshold quorum ($|Quorum| = Q_C + b$). The fourth column gives the quorum load of each storage-node (i.e., the fraction of operations for which a storage-node must execute a request). Lastly, the fifth column shows the calculated system throughput normalized to the throughput of $\Delta = 0$.*

with. Note, that as Δ increases, the ratio of the size of the threshold quorum to N decreases. For threshold quorums to work without requiring frequent repair or object syncing, it is necessary for all clients accessing a data-item to interact with that data-item through the same quorum. If many distinct quorums are used to update/or query a data-item, repair and/or object syncing will be necessary (this is not always the case if all storage-nodes are always updated—as is the default in all other experiments where $\Delta = 0$). The fourth column shows the expected load of a single storage-node (i.e., $\frac{quorum\_size}{N}$). Column five shows the expected system throughput normalized to $\Delta = 0$.

Figure 5.11 shows the throughput of the PMD service when using a threshold quorum construction, as Δ increases. The throughput is normalized to the throughput obtained at $\Delta = 0$. Two curves are plotted on the graph. The first line shows the calculated throughput (see column 5 in Table 5.3). The second line shows the maximum throughput attained by running a heavy weight synthetic update operation containing a 4 KB argument and a 10 ms storage-node think time.

To measure the throughput, many clients, each with multiple outstanding queries or updates are employed. The reported throughput measurements are for a saturated system (i.e., adding more clients does not increase throughput) We employ an access strategy based on a deterministic function of the object ID. Such an access strategy results in updates for a given object preferentially accessing the same quorum (i.e., the *preferred*

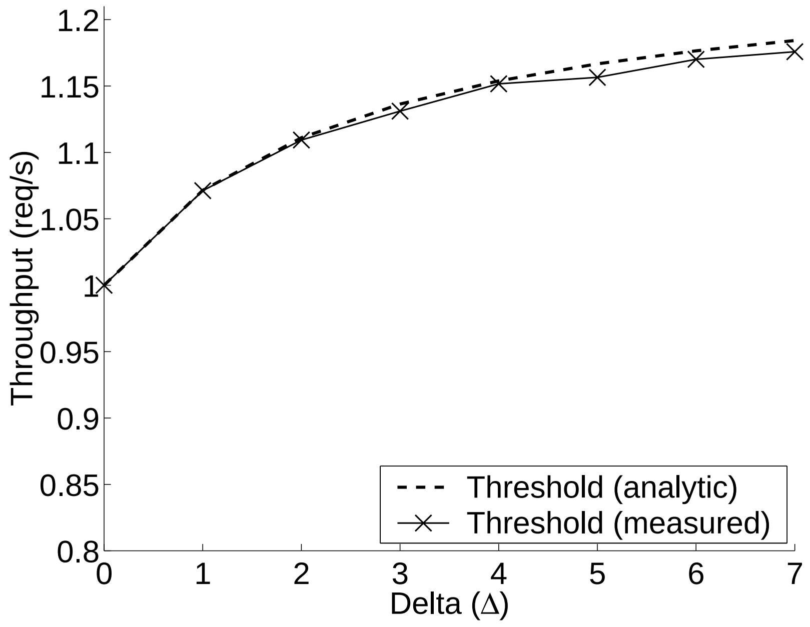


Figure 5.11: **Throughput of threshold quorum system vs. Δ.** *This graph shows the normalized system throughput as Δ is increased. Throughput is normalized to that of* $\Delta = 0$. *See Table 5.3 for a description of the relationship between Δ, N, and throughput. Two curves are shown. The first line shows the calculated throughput as described in Table 5.3. The second line shows the throughput attained when running a 4KB update operation.*

quorum). Accessing a service that is implemented by an ensemble of Q/U objects, via each objects preferred quorum, approximates a traditional quorum access strategy. As can be seen, the synthetic update line closely follows the calculated throughput curve.

An additional experiment was run using a recursive threshold construction [Malkhi et al. 1997]. For the recursive threshold construction (with $t = b$), $N = (5b+1)^{\Delta+1}$ and $|Quorum| = (4b+1)^{\Delta+1}$ (i.e., Δ indicates recursion depth). With $t = b = 1$ ($\Delta = 1, N = 36$, $|Quorum| = 25$), the achieved throughput, normalized to $\Delta = 0$, was 1.19 versus 1.20 for the normalized theoretical throughput.

5.6.5 PASIS file system: SSH-build

The `SSH-build` benchmark was constructed as a replacement for the Andrew file system benchmark [Howard et al. 1988]. This experiment also demonstrates the effect of

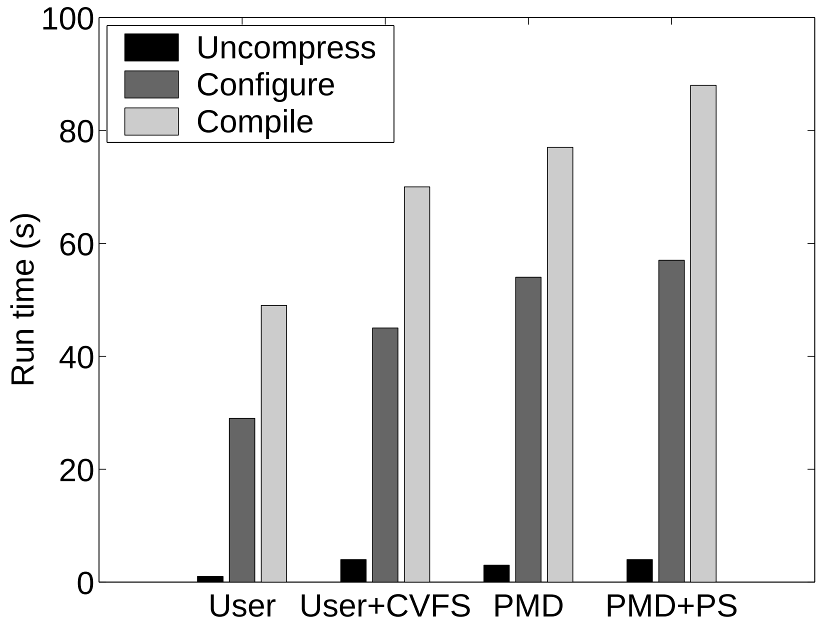


Figure 5.12: **Run time of SSH build for Linux, PMD, and PMD + PS.**

the PMD service operating in unison with the PS service. This benchmark consists of three phases: unpacking the OpenSSH archive, running `configure`, and compiling the OpenSSH binaries. The unpack phase stresses metadata operations on files of varying sizes by uncompressing and untaring the OpenSSH (v3.8p1) tar archive. The configure phase consists of the automatic generation of header files and Makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables. Figure 5.12 shows the runtime of the `SSH-build` benchmark for four configurations.

None of the NFS configurations use the synchronous mount option. However, all NFS configurations use a 128 MB write-back cache with no data expiration time. The first set of bars show a user-level NFSv3 server that stores files in the local ext3 file system. The second set of bars show the performance of the same user-level NFSv3 server just described, however the data is stored across the network to a single CVFS storage-node. This shows the overhead of using CVFS across an additional network link. However, in this configuration metadata is treated as data (i.e., attributes and directory entries are stored within data blocks). The third set of bars show the cost of using the PASIS metadata service (with $t = b = 1, N = 6$) storing file data in the local file system. The fourth, and last, set of bars show the `SSH-build` benchmark run against the complete PASIS metadata and storage service. Both the metadata and storage service are configured with $t = b = 1$; $N = 6$ for the metadata service and $N = 5$ for the storage service. As can be seen there is almost a 2x performance difference between an user-level NFS server with no fault-tolerance and an NFS server backed by a Byzantine fault-tolerant metadata and storage service that provides strong consistency. The majority of the overhead is due to the extra communication required (11 storage-nodes in the PMD+PS case vs. 1 without), as well there is a non-zero cost in our CVFS and storage-node implementations.

5.7 Summary

This chapter describes the PASIS metadata (PMD) service. It uses a novel quorum-style query/update (Q/U) protocol to provide horizontal scalability for metadata, as is enjoyed for data in scalable storage systems. The PMD service extends the read/conditional write protocol, described in Chapter 4, to support more general query and update operations. These operations provide access to objects at a finer-granularity than do block-based protocols (e.g., reading/inserting directory entries vs. reading/writing full directories). In addition, atomic updates across multiple objects are supported.

Similar to the other protocols developed thus far, the Q/U protocol uses optimism and versioning to achieve efficiency while tolerating asynchronous communications and Byzantine failures of clients and servers. Experiments with a decentralized NFS file service demonstrate feasibility and efficiency. As well, performance under concurrency and faults is examined. Experiments also show that threshold quorum constructions can be used to significantly increase throughput without requiring the partitioning of the metadata service.

6 Conclusions and Future Work

6.1 Conclusion

This thesis has demonstrated a novel approach to achieving scalable, highly fault-tolerant storage systems by leveraging a set of efficient and scalable, strong consistency protocols enabled by storage-node versioning. These consistency protocols achieve efficiency and scalability via a combination of optimistic operation, versioning, and quorum-style redundancy.

Three consistency protocols have been developed that offer varying semantics useful for building different components within a survivable, decentralized storage-system. The first protocol, the read/write protocol (R/W), provides read–write semantics of full data blocks. This protocol is suitable as the basis for the data storage component within a survivable storage system, since most block based data services expect whole block updates.

The second protocol, the read/conditional-write (R/CW) protocol, provides read–modify–write semantics of full data blocks. While this protocol also assumes blocks (or data objects) are read and written as atomic units, it offers stronger consistency guarantees. These semantics guarantee that the data region has not been modified between a read and a successive write operation to the same data region.

The third protocol, the query/update (Q/U) protocol, extends the R/CW protocol to more fully support the semantics required by metdata. In order to preserve the consistency of metadata, metadata objects (e.g., directories) require update operations that modify existing contents (such as inserting a new directory entry), rather than overwriting their previous contents. As well, metadata usually requires atomic update operations across multiple metadata objects (e.g., when performing a rename, or moving files). The Q/U protocol provides for the serializability of multiple, arbitrary operations through the use of replicated state machines.

These protocols were developed in detail, evaluated individually, and used as a basis for building a fault-tolerant, scalable storage-system. Results show that the PASIS file system configured to tolerate one Byzantine fault is within a factor of two in the response time unpacking, configuring, and building OpenSSH as compared to an unreplicated user-level NFS server. The storage service component of the file system, using the R/W protocol, was shown to scale well in terms of both throughput and response time as number of faults tolerated is scaled up. As well, it performs well when compared to a Byzantine fault-tolerant agreement protocol (BFT) and by offloading work from storage-nodes to clients increases its scalability. Results also show that the PASIS metadata service, using the query/update protocol, scales with as the number clients is increased and reponse time increases slightly as the number of faults tolerated is scaled up. Additionally, the use of quorum thresholds enables the system's throughput to scale close to its theoretical bounds and it is expected that other quorum constructions can further increase the system's scalability.

6.2 Contributions

This main contribution of this thesis is the design and evaluation of three consistency protocols that have been enabled by versioning storage-nodes. These contributions are:

(1) The development and demonstration of a read/write block storage consistency protocol that enables highly fault-tolerant storage through the use of erasure coded data and versioning storage-nodes. Its correctness is shown through proof sketches.

(2) The development and demonstration of a read/conditional-write block protocol that allows for stronger read–modify–write consistency semantics. Additionally, trade-

offs between tolerating Byzantine clients and erasure coding, as well as tradeoffs between tolerating Byzantine storage-nodes and liveness have been discussed.

(3) The extention of the read/conditional-write block protocol (in the query/update protocol) to support operations on multiple, arbitrary objects and the implementation of a scalable metadata service based upon the query/update and the read/write protocol.

(4) The evaluation of a distributed file system that utilizes the scalability and fault-tolerance of the developed consistency protocols in terms of the number of faults tolerated, the maximum throughput the system can sustain, and its performance in degraded operation modes (i.e., with concurrency and faults).

6.3 Future directions

While this thesis has demonstrated the feasibility of using versioning storage-nodes to provide consistency through the use of scalable, optimistic protocols, there are many tradeoffs and design decisions that remain unanswered.

There exist additional system models that use stronger assumptions to reduce the constraints (in terms of N, Q_C, and m) of the protocol in use. For example, we have developed a family of R/W protocols that enable the client to choose between a synchronous or asynchronous timing model, Byzantine or crash fault models (for both clients and storage-nodes), and repair or non-repair [Goodson et al. 2003]. There is a tradeoff between when to use which of the protocol family members. Additionally, alternative fault models could be examined. The focus, in the work so far, has been on Byzantine and crash faults. Additional fault models between these two extremes exist. For example, many applications may not require the expensive cost of Byzantine faults, but require protection from integrity or value faults, which may lead to lower constraints. As well, non-colluding Byzantine faults may be considered.

For the block-based protocols there is the tradeoff between when to use the different encoding schemes. There exists the option to increase space-efficiency by increasing the *m* parameter. However, increasing *m* also requires an increase in *N* and Q_C, which requires communicating with a larger number of storage-nodes and thus leads to higher disk head utilization. Different workloads may require different configurations and encoding schemes (e.g., the use of replication over the striping of data, or the combination of replication and erasure coding). Identifying the correct configuration to use for each application is an interesting problem. Similarly, different quorum constructions have different properties in terms of load and scalability. Thus, knowing when to transition between different quorum constructions could prove to be of great benefit to the system. Additionally, an examination of the availability and reliability of the system when using different encoding schemes can be made.

There are also a number of protocol optimizations that can be used. For example, authenticators could be used the R/W protocol to enable garbage collection without the need for storage-nodes to communicate. Additionally, write witnesses may be used to increase space-efficiency in many of the protocols. Similar to read witnesses, write witnesses hold only a timestamp with no data, but have the ability to vote for a specific piece of data by use of the data hash contained within the timestamp.

In terms of the file system, a number of tradeoffs and open questions exist. In order to provide true file sharing between clients, the issue of client cache coherency must be addressed. There are traditional methods such as locks and leases with (or without) callbacks that solve this problem. However, synchrony assumptions are often introduced to detect clients who fail while holding locks. The impact of these assumptions have not been examined. As well, a fully functional file system requires the enforcement of access control. While an overview of lock/lease and access control objects was discussed, they have not been implemented. Lastly, a thorough examination of the tradeoffs involved in desiging a storage-service can be made (e.g., the use of the R/W protocol vs. the R/CW protocol in providing the storage-service).

Bibliography

ABD-EL-MALEK, M., GANGER, G. R., GOODSON, G. R., WYLIE, J. J., AND REITER, M. K. 2004. The safety and liveness properties of the Read/Conditional-Write protocol and Query/Update protocol. Tech. Rep. CMU–PDL–04–107, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA.

ABRAHAM, I., CHOCKLER, G., KEIDAR, I., AND MALKHI, D. 2004. Byzantine Disk Paxos: optimal resilience with Byzantine shared memory. In *ACM Symposium on Principles of Distributed Computing*. ACM.

ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. 2002. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. In *Symposium on Operating Systems Design and Implementation*. USENIX Association, 1–15.

AGRAWAL, D. AND EL ABBADI, A. 1990. Integrating security with fault-tolerant distributed databases. *Computer Journal 33,* 1, 71–78.

AGUILERA, M. K., CHEN, W., AND TOUEG, S. 2000. Failure detection and consensus in the crash-recovery model. *Distributed Computing 13,* 2, 99–125.

AGUILERA, M. K. AND FROLUND, S. 2003. Strict linearizability and the power of aborting. Tech. Rep. HPL-2003-241, HP Labs.

AMIRI, K., GIBSON, G. A., AND GOLDING, R. 1999. Scalable concurrency control and recovery for shared storage arrays. Tech. Rep. CMU–CS–99–111, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.

AMIRI, K., GIBSON, G. A., AND GOLDING, R. 2000a. Highly concurrent shared storage. In *International Conference on Distributed Computing Systems*. IEEE Computer Society, 298–307.

AMIRI, K., GIBSON, G. A., AND GOLDING, R. 2000b. Highly concurrent shared storage. In *International Conference on Distributed Computing Systems*. IEEE, 298–397.

ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. 1996. Serverless network file systems. *ACM Transactions on Computer Systems 14,* 1, 41–79.

BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. 1991. Measurements of a distributed file system. In *ACM Symposium on Operating System Principles*. 198–212.

BERLEKAMP, E. 1968. *Algebraic coding theory*. McGraw-Hill, New York.

BERNSTEIN, P. A., SHIPMAN, D. W., AND ROTHNIE JR, J. B. 1980. Concurrency control in a system for distributed databases. *ACM Transactions on Database Systems 5,* 1, 18–51.

BLACKWELL, T., HARRIS, J., AND SELTZER, M. 1995. Heuristic cleaning algorithms in log-structured file systems. In *USENIX Annual Technical Conference*. USENIX Association, 277–288.

BRAAM, P. J. 2004. *The Lustre storage architecture*. Cluster File Systems, Inc.

BRACHA, G. AND TOUEG, S. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM 32,* 4, 824–840.

CABRERA, L.-F. AND LONG, D. D. E. 1991. Swift: using distributed disk striping to provide high I/O data rates. *Computing Systems 4,* 4, 405–436.

CACHIN, C., KURSAWE, K., PETZOLD, F., AND SHOUP, V. 2001. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology - CRYPTO*. IEEE, 524–541.

CASTRO, M. AND LISKOV, B. 1998a. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*. ACM, 173–186.

CASTRO, M. AND LISKOV, B. 1998b. Practical byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*. ACM, 173–186.

CASTRO, M. AND LISKOV, B. 2001. Byzantine fault tolerance can be fast. In *Dependable Systems and Networks*. 513–518.

CASTRO, M. AND LISKOV, B. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20,* 4, 398–461.

CASTRO, M. AND RODRIGUES, R. 2003. *BFT library implementation*. http://www.pmg.lcs.mit.edu/bft/#sw.

CHANG, F., JI, M., LEUNG, S.-T. A., MACCORMICK, J., PERL, S., AND ZHANG, L. 2002. Myriad: cost-effective disaster tolerance. In *Conference on File and Storage Technologies*. USENIX Association, 103–116.

CHOR, B., GOLDWASSER, S., MICALI, S., AND AWERBUCH, B. 1985. Verifiable Secret Sharing in the Presence of Faults. In *IEEE Symposium on Foundations of Computer Science*. IEEE, 335–344.

CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. 1995. Atomic broadcast: from simple message diffusion to Byzantine agreement. *Information and Computation 118,* 1, 158–179.

DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. 2001. Wide-area cooperative storage with CFS. In *ACM Symposium on Operating System Principles*. 202–215.

DAI, W. 2003. *Crypto++ reference manual*. http://cryptopp.sourceforge.net/docs/ref/.

DESWARTE, Y., BLAIN, L., AND FABRE, J.-C. 1991. Intrusion tolerance in distributed computing systems. In *IEEE Symposium on Security and Privacy*. 110–121.

DWORK, C., LYNCH, N., AND STOCKMEYER, L. 1988. Consensus in the presence of partial synchrony. *Journal of the Association for Computing Machinery 35,* 2, 288–323.

FELDMAN, P. 1987. A practical scheme for non-interactive verifiable secret sharing. In *IEEE Symposium on Foundations of Computer Science*. IEEE, 427–437.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32,* 2, 374–382.

FRØLUND, S., MERCHANT, A., SAITO, Y., SPENCE, S., AND VEITCH, A. 2004. A decentralized algorithm for erasure-coded virtual disks. In *International Conference on Dependable Systems and Networks*. IEEE, 125–134.

GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. 1998. A cost-effective, high-bandwidth storage architecture. In *Architectural Support for Programming Languages and Operating Systems*. 92–103.

GIFFORD, D. K. 1979. Weighted voting for replicated data. In *ACM Symposium on Operating System Principles*. ACM, 150–162.

GOLDING, R., BOSCH, P., STAELIN, C., SULLIVAN, T., AND WILKES, J. 1995. Idleness is not sloth. In *Winter USENIX Technical Conference*. USENIX Association, 201–212.

GONG, L. 1989. Securely replicating authentication services. In *International Conference on Distributed Computing Systems*. IEEE Computer Society Press, 85–91.

GOODSON, G. R., WYLIE, J. J., GANGER, G. R., AND REITER, M. K. 2003. A protocol family for versatile survivable storage infrastructures. Tech. Rep. CMU-PDL-03-103, CMU.

GRAY, C. G. AND CHERITON, D. R. 1989. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *ACM Symposium on Operating System Principles*. 202–210.

GRAY, J. N. 1978. *Notes on data base operating systems*. Vol. 60. Springer-Verlag, Berlin, 393–481.

GRAY, J. N., LORIE, R. A., PUTZOLU, G. R., AND TRAIGER, I. L. 1976. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*. IFIP, 365–394.

HAERDER, T. AND REUTER, A. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys 15,* 4, 287–317.

HERLIHY, M. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages 13,* 1, 124–149.

HERLIHY, M. P. AND TYGAR, J. D. 1987. How to make replicated data secure. In *Advances in Cryptology - CRYPTO*. Springer-Verlag, 379–391.

HERLIHY, M. P. AND WING, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12,* 3, 463–492.

HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. 1988. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS) 6,* 1, 51–81.

JAYANTI, P., CHANDRA, T. D., AND TOUEG, S. 1998. Fault-tolerant wait-free shared objects. *Journal of the ACM 45,* 3, 451–500.

KATCHER, J. 1997. PostMark: a new file system benchmark. Tech. Rep. TR3022, Network Appliance.

KIHLSTROM, K. P., MOSER, L. E., AND MELLIAR-SMITH, P. M. 2001. The SecureRing group communication system. *ACM Transactions on Information and Systems Security 1,* 4, 371–406.

KRAWCZYK, H. 1994. Secret sharing made short. In *Advances in Cryptology - CRYPTO*. Springer-Verlag, 136–146.

KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. 1988. Efficient synchronization of multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems 10,* 4, 579–601.

KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATEN, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. 2000. OceanStore: an architecture for global-scale persistent storage. In *Architectural Support for Programming Languages and Operating Systems*. 190–201.

KUNG, H. T. AND ROBINSON, J. T. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6,* 2, 213–226.

LAMPORT, L. 1998. The part-time parliament. *ACM Transactions on Computer Systems 16,* 2, 133–169.

LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4,* 3, 382–401.

LEE, E. K. AND THEKKATH, C. A. 1996. Petal: distributed virtual disks. In *Architectural Support for Programming Languages and Operating Systems*. 84–92.

LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. 1991. Replication in the Harp file system. In *ACM Symposium on Operating System Principles*. 226–238.

LONG, D. D. E., MONTAGUE, B. R., AND CABRERA, L.-F. 1994. Swift/RAID: a distributed RAID system. *Computing Systems 7,* 3, 333–359.

LUBY, M. G., MITZENMACHER, M., SHOKROLLAHI, M. A., AND SPIELMAN., D. A. 2001. Efficient Erasure Correcting Codes. *IEEE Transactions on Information Theory 47,* 2, 569–584.

MALKHI, D. AND REITER, M. 1997. Byzantine quorum systems. In *ACM Symposium on Theory of Computing*. ACM, 569–578.

MALKHI, D. AND REITER, M. 1998a. Byzantine quorum systems. *Distributed Computing 11*, 4, 203–213.

MALKHI, D., REITER, M., AND WOOL, A. 1997. The load and availability of Byzantine quorum systems. In *ACM Symposium on Principles of Distributed Computing*. ACM, 249–257.

MALKHI, D., REITER, M., AND WOOL, A. 2000. The load and availability of byzantine quorum systems. *SIAM Journal of Computing 29,* 6, 1889–1906.

MALKHI, D. AND REITER, M. K. 1998b. Secure and scalable replication in Phalanx. In *IEEE Symposium on Reliable Distributed Networks*.

MALKHI, D., REITER, M. K., TULONE, D., AND ZISKIND, E. 2001. Persistent objects in the Fleet system. In *DARPA Information Survivability Conference and Exposition*. IEEE, 126–136.

MARTIN, J.-P., ALVISI, L., AND DAHLIN, M. 2002. Minimal Byzantine storage. In *International Symposium on Distributed Computing*.

MENON, J., APEASE, D., REES, R., DUYANOVICH, L., AND HILLSBERG, B. 2003. IBM Storage Tank - A heterogeneous scalable SAN file system. *IBM Systems Journal 42,* 2, 250–267.

MERKLE, R. C. 1987. A digital signature based on a conventional encryption function. In *Advances in Cryptology - CRYPTO*.

MUKKAMALA, R. 1994. Storage efficient and secure replicated distributed databases. *IEEE Transactions on Knowledge and Data Engineering 6,* 2, 337–341.

MULLENDER, S. J. 1985. A distributed file service based on optimistic concurrency control. In *ACM Symposium on Operating System Principles*. 51–62.

MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. 2001. A low-bandwidth network file system. In *ACM Symposium on Operating System Principles*. ACM, 174–187.

MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. 2002. Ivy: a read/write peer-to-peer file system. In *Symposium on Operating Systems Design and Implementation*. USENIX Association.

NAOR, M. AND WOOL, A. 1998. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing 27,* 2, 423–447.

NOBLE, B. D. AND SATYANARAYANAN, M. 1994. An empirical study of a highly available file system. Tech. Rep. CMU–CS–94–120, Carnegie Mellon University.

PAPADIMITRIOU, C. H. 1979. The serializability of concurrent database updates. *Journal of the ACM 26,* 4, 631–653.

PÂRIS, J.-F. 1986. Voting with witnesses: a consistency scheme for replicated files. In *International Conference on Distributed Computing Systems*. IEEE Computer Society, 606–612.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD International Conference on Management of Data*. 109–116.

PENNINGTON, A. G., STRUNK, J. D., GRIFFIN, J. L., SOULES, C. A. N., GOODSON, G. R., AND GANGER, G. R. 2003. Storage-based intrusion detection: watching storage activity for suspicious behavior. In *USENIX Security Symposium*.

PIERCE, E. T. 2001. Self-adjusting quorum systems for byzantine fault tolerance. Ph.D. thesis, Department of Computer Sciences, University of Texas at Austin.

PITTELLI, F. M. AND GARCIA-MOLINA, H. 1989. Reliable scheduling in a TMR database system. *ACM Transactions on Computer Systems 7,* 1, 25–60.

RABIN, M. O. 1989. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36,* 2, 335–348.

REED, D. P. 1983. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems 1,* 1, 3–23.

REED, D. P. AND SVOBODOVA, L. 1980. SWALLOW: a distributed data storage system for a local network. In *International Workshop on Local Networks*.

REITER, M. K. AND BIRMAN, K. P. 1994. How to securely replicate services. *ACM Transactions on Programming Languages and Systems 16,* 3, 986–1009.

RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. 2003. Pond: the OceanStore prototype. In *Conference on File and Storage Technologies*. USENIX Association.

RIVEST, R. 1992. *RFC1321: The MD5 Message-Digest Algorithm*. http://www.ietf.org/rfc/rfc1321.txt.

ROWSTRON, A. AND DRUSCHEL, P. 2001. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *ACM Symposium on Operating System Principles*. ACM, 188–201.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22,* 4, 299–319.

SHAMIR, A. 1979. How to share a secret. *Communications of the ACM 22,* 11, 612–613.

SHRIVASTAVA, S. K., EZHILCHELVAN, P. D., SPEIRS, N. A., AND TULLY, A. 1992. Principal features of the voltan family of reliable node architectures for distributed systems. *IEEE Transactions on Computr 41*, 5, 542–549.

SKEEN, D. AND STONEBRAKER, M. 1983. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering 9*, 3, 261–270.

SOLTIS, S. R., RUWART, T. M., AND O'KEEFE, M. T. 1996. The global file system. In *NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*. IEEE Computer Society Press.

SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. 2003. Metadata efficiency in versioning file systems. In *Conference on File and Storage Technologies*. USENIX Association, 43–58.

STEINER, J. G., SCHILLER, J. I., AND NEUMAN, C. 1988. Kerberos: an authentication service for open network systems. In *Winter USENIX Technical Conference*. 191–202.

STRUNK, J. D., GOODSON, G. R., PENNINGTON, A. G., SOULES, C. A. N., AND GANGER, G. R. 2002. Intrusion detection, diagnosis, and recovery with self-securing storage. Tech. Rep. CMU–CS–02–140, Carnegie Mellon University.

STRUNK, J. D., GOODSON, G. R., SCHEINHOLTZ, M. L., SOULES, C. A. N., AND GANGER, G. R. 2000. Self-securing storage: protecting data in compromised systems. In *Symposium on Operating Systems Design and Implementation*. USENIX Association, 165–180.

THAMBIDURAI, P. AND PARK, Y.-K. 1988. Interactive consistency with multiple failure modes. In *Symposium on Reliable Distributed Systems*. IEEE, 93–100.

THEKKATH, C. A., MANN, T., AND LEE, E. K. 1997. Frangipani: a scalable distributed file system. In *ACM Symposium on Operating System Principles*. ACM, 224–237.

THOMAS, R. H. 1979. A majority consensus approach to concurrency control. *ACM Transactions on Database Systems 4*, 180–209.

WEATHERSPOON, H. AND KUBIATOWICZ, J. D. 2002. Erasure coding vs. replication: a quantitative approach. In *First International Workshop on Peer-to-Peer Systems (IPTPS 2002)*.

WYLIE, J. J., BIGRIGG, M. W., STRUNK, J. D., GANGER, G. R., KILICCOTE, H., AND KHOSLA, P. K. 2000. Survivable information storage systems. *IEEE Computer 33,* 8, 61–68.

A Read/write safety and liveness

A.1 Proof of safety

This section presents a proof that our protocol implements linearizability [Herlihy and Wing 1990] as adapted appropriately for a fault model admitting operations by Byzantine clients.

A.1.1 Safety guarantees

Intuitively, linearizability requires that each read operation return a value consistent with some execution in which each read and write is performed at a distinct point in time between when the client invokes the operation and when the operation returns. The adaptations necessary to reasonably interpret linearizability in our context arise from the fact that Byzantine clients need not follow the read and write protocols and that read operations may abort in non-repair member protocols. We consider four distinct safety guarantees:

Linearizability

Repairable protocol members with crash-only clients achieve linearizability as originally defined by Herlihy and Wing [Herlihy and Wing 1990].

Byzantine-operation linearizability

Read operations by Byzantine clients are excluded from the set of linearizable operations. Write operations are only included if they are well-formed (i.e., if they are single-valued as in Section 3.2).

Write operations by Byzantine clients do not have a well-defined start time. Such operations are concurrent to all operations that begin before they complete and to all operations that are also performed by Byzantine clients. A Byzantine client can write "back in time" by using a lower logical timestamp than a benign client would have used. Since write operations by Byzantine clients are concurrent to all operations that started before it completed, they can be linearized just prior to some concurrent write operation (if there is one). Such a linearization ensures that the Byzantine "back in time" write operation has no effect since the value written is never returned by a read operation.

In summary, there are two types of Byzantine write operations that are of concern: writes that are not well-formed and "back in time" writes. In the case that the Byzantine write operation is not well-formed, read operations by benign clients exclude it from the set of linearized operations. In the case that the Byzantine write operation is "back in time", the protocol family achieves something similar, in that such Byzantine write operations are linearized so that they have no effect.

A.1.2 Proof

Because return values of reads by Byzantine clients obviously need not comply with any correctness criteria, we disregard read operations by Byzantine clients in reasoning about linearizability, and define the duration of reads only for those executed by benign clients only.

DEFINITION 1 A read operation executed by a benign client *begins* when the client invokes `READ` locally. A read operation executed by a benign client *completes* when this invocation returns $\langle \mathit{timestamp}, \mathit{value} \rangle$. A read operation by a benign client that crashes before the read completes, does not complete.

Before defining the duration of write operations, it is necessary to define what it means for a storage-node to *accept* and then *execute* a write request.

DEFINITION 2 Storage-node S, *accepts* a write request with data-fragment D, cross checksum CC, and timestamp ts upon successful return of the function `VALIDATE_WRITE`(ts, D, CC) at the storage-node.

DEFINITION 3 Storage-node S, *executes* a write request once the write request is accepted. An executed write request is stored in stable storage.

It is not well defined when a write operation by a Byzantine client begins. Therefore, we settle for merely a definition of when writes by Byzantine clients complete.

DEFINITION 4 A write operation with timestamp *ts completes* once Q_C benign storage-nodes have executed write requests with timestamp ts.

In fact, Definition 4 applies to write operations by benign clients as well as "write operations" by Byzantine clients. In this section, we use the label w_{ts} as a shorthand for the write operation with timestamp ts. In contrast to Definition 4, Definition 5 applies only to write operations by benign clients.

DEFINITION 5 w_{ts} *begins* when a benign client invokes the `WRITE` operation locally that issues a write request bearing timestamp ts.

LEMMA 1 *Let c_1 and c_2 be benign clients. If c_1 performs a read operation that returns $\langle ts_1, v_1\rangle$, c_2 performs a read operation that returns $\langle ts_2, v_2\rangle$, and $ts_1 = ts_2$, then $v_1 = v_2$.*

Proof: Since $ts_1 = ts_2$, each read operation considers the same verifier. Since each read operation considers the same verifier, each read operation considers the same cross checksum (remember, a collision resistant hash function is employed). A read operation does not return a value unless the cross checksum is valid and there are more than b read responses with the timestamp (since only candidates classified as repairable or complete are considered). Thus, only a set of data-fragments resulting from the erasure-coding of the same data-item that are issued as write requests with the same timestamp can validate a cross checksum. As such, v_1 and v_2 must be the same. $\square$

Let v_{ts} denote the value written by w_{ts} which, by Lemma 1, is well-defined. We use r_{ts} to denote a read operation by a benign client that returns $\langle ts, v_{ts} \rangle$.

DEFINITION 6 Let o_1 denote an operation that completes (a read operation by a benign client, or a write operation), and let o_2 denote an operation that begins (a read or write by a benign client). o_1 *precedes* o_2 if o_1 completes before o_2 begins. The precedence relation is written as $o_1 \rightarrow o_2$. Operation o_2 is said to follow, or to be subsequent to, operation o_1.

LEMMA 2 *If $w_{ts} \rightarrow w_{ts'}$, then $ts < ts'$.*

Proof: A complete write operation executes at at least Q_C benign storage-nodes (cf. Definition 4). Since $w_{ts} \rightarrow w_{ts'}$, the `READ_TIMESTAMP` function for w_{ts} collects $N-t$ `TIME_RESPONSE` messages, and so $w_{ts'}$ observes at least $b+1$ `TIME_RESPONSE` messages from benign storage-nodes that executed w_{ts} (remember, $t+b < Q_C$ for all asynchronous protocol family members). As such, $w_{ts'}$ observes some timestamp greater than or equal to ts and constructs ts' to be greater than ts. A Byzantine storage-node can return a logical timestamp greater than that of the preceding write operation; however, this still advances logical time and Lemma 2 holds. □

OBSERVATION 1 Timestamp order is a total order on write operations. The timestamps of write operations by benign clients respect the precedence order among writes.

LEMMA 3 *If some read operation by a benign client returns $\langle ts, v_{ts} \rangle$, with $v_{ts} \neq \perp$, then w_{ts} is complete.*

Proof: For a read operation to return value v_{ts}, the value must have been observed at at least $Q_C + b$ storage-nodes (given the complete classification rule for candidate sets). Since, at most b storage-nodes are Byzantine, the write operation w_{ts} has been executed by at least Q_C benign storage-nodes. By definition, w_{ts} is complete. □

OBSERVATION 2 The read operation from Lemma 3 could have performed repair before returning. In a repairable protocol member, a candidate that is neither classifiable as incomplete or complete is repaired. Once repaired, the candidate is complete.

DEFINITION 7 w_{ts} is *well-formed* if $ts.Verifier$ equals the hash of cross checksum CC, and for all $i \in \{1, \ldots, N\}$, hash $CC[i]$ of the cross checksum equals the hash of data-fragment i that results from the erasure-encoding of v_{ts}.

LEMMA 4 *If w_{ts} is well-formed, and if $w_{ts} \rightarrow r_{ts'}$, then $ts \leq ts'$.*

Proof: Since w_{ts} is well-formed it can be returned by a read operation. By Lemma 3, read operations only return values from complete write operations. As such, $r_{ts'}$ must either return the value with timestamp ts or a value with a greater timestamp. Therefore, $ts \leq ts'$. □

OBSERVATION 3 It follows from Lemma 4 that for any read r_{ts}, either $w_{ts} \rightarrow r_{ts}$ and w_{ts} is the latest complete write that precedes r_{ts}, or $w_{ts} \not\rightarrow r_{ts}$ and $r_{ts} \not\rightarrow w_{ts}$ (i.e., w_{ts} and r_{ts} are concurrent).

OBSERVATION 4 It also follows from Lemmas 3 and 4 that if $r_{ts} \rightarrow r_{ts'}$, then $ts \leq ts'$. As such, there is a partial order $\prec$ on read operations by benign clients defined by the timestamps associated with the values returned (i.e., of the write operations read). More formally, $r_{ts} \prec r_{ts'} \Longleftrightarrow ts < ts'$.

Since Lemma 2 ensures a total order on write operations, ordering reads according to the timestamps of the write operations whose values they return yields a partial order on read operations. Lemma 4 ensures that this partial order is consistent with precedence among reads. Therefore, any way of extending this partial order to a total order yields an ordering of reads that is consistent with precedence among reads. Thus, Lemmas 2 and 4 guarantee that this totally ordered set of operations is consistent with precedence. This implies the natural extension of linearizability to our fault model (i.e., ignoring reads by Byzantine clients and the begin time of writes by Byzantine clients); in particular, it implies linearizability as originally defined by Herlihy [Herlihy and Wing 1990] for the read/write protocol if all clients are benign.

A.2 Proof of liveness

This section presents the proof of the liveness properties of protocol members.

A.2.1 Liveness guarantees

There are two distinct liveness guarantees: wait-freedom and single-client wait-freedom. These guarantees hold so long as the storage capacity on storage-nodes is not exhausted.

Wait-freedom

Wait-freedom is a desirable liveness property [Herlihy 1991]. Informally, achieving wait-freedom means that each client can complete its operations in finitely many steps regardless of the actions performed or failures experienced by other clients. For a formal definitions see [Herlihy 1991].

Unbounded storage capacity

In the proof of liveness for read operations, we assume that storage-nodes have unbounded storage capacity (i.e., that the entire version history back to the initial value $\perp$ at time $\mathbf{0}$ is available at each storage-node). To prevent capacity exhaustion, some garbage collection mechanism is required. Garbage collection reduces the liveness of read operations. A read operation that is concurrent to write operations and to garbage collection may not observe a complete candidate. The read operation can observe a series of incomplete candidates that complete and are garbage collected within the duration of the read operation. In such a situation, the read operation would observe $\perp$ at some timestamp other than $\mathbf{0}$ from storage-nodes, indicating that the client has "skipped" over a complete write operation. The read operation then must be retried. The implementation details of garbage collection and its impact on liveness properties is given in Section 3.7.3.

A.2.2 Proof

All liveness properties hinge on the following lemma.

LEMMA 5 *All operations eventually receive at least $N-t$ responses.*

Proof: In the crash-recovery model, there are at least $N-t$ good storage-nodes (i.e., storage-nodes that are always-up or eventually-up). By definition, eventually, all good storage-nodes will be up. Since all requests to storage-nodes, from clients, are retried until $N-t$ responses are received, eventually, $N-t$ responses will be received (see READ_TIMESTAMP, DO_WRITE, and DO_READ). □

OBSERVATION 5 It is possible for progress to be made throughout the duration of a run, not just once all good storage-nodes are up. Lemma 5 guarantees that eventually $N-t$ responses will be received. During any period in which $N-t$ storage-nodes are up, operations may receive $N-t$ responses and thus complete. In fact, responses can be collected, over time, from $N-t$ storage-nodes, during a period in which fewer than $N-t$ storage-nodes are ever up (but during which some storage-nodes crash and some recover).

Asynchronous repairable

The asynchronous repairable protocol member provides a strong liveness property, namely wait-freedom [Herlihy 1991; Jayanti et al. 1998]. Informally, each operation by a correct client completes with certainty, even if all other clients fail, provided that at most b servers suffer Byzantine failures and no more than t servers are not good.

LEMMA 6 *A write operation by a correct client completes.*

Proof: A write operation by a correct client waits for $N-t$ responses from storage-nodes before returning. By Lemma 5, $N-t$ responses can always be collected. Since, $Q_C \leq N-t-b$ (cf. (3.5) in Section 3.4) for repairable protocol members, then $N-t \geq Q_C + b$. Since at most b storage-nodes are Byzantine, then at least Q_C benign storage-nodes execute write requests, which completes the write operation. □

LEMMA 7 *A read operation by a correct client completes.*

Proof: Given $N - t$ READ_RESPONSE messages, a read operation classifies a candidate as complete, repairable, or incomplete. The read completes if a candidate is classified as complete. As well, the read completes if a candidate is repairable. Repair is initiated for repairable candidates—repair performs a write operation, which by Lemma 6 completes—which lets the read operation complete. In the case of an incomplete, the read operation traverses the version history backwards, until a complete or repairable candidate is discovered. Traversal of the version history terminates if $\perp$ at logical time $\mathbf{0}$ is encountered at Q_C storage-nodes. $\square$